尚硅谷Java技术之高频面试题

版本：V4.0

尚硅谷Java辅导部

# 第一章 面试技巧篇

## 1、面试过程最关键的是什么？

1）不是你说了什么，而是你怎么说

2）大大方方的聊，放松，总结经验就是一点---淡定，越淡定越真实，回答的越好

## 2、面试时该怎么说？

1）语言表达清楚

（1）思维逻辑清晰，表达流畅

（2）一二三层次表达

2）所述内容不犯错

（1）不说前东家或者自己的坏话

（2）说自己擅长的方面，重点就是---遇到问题，往自己会的方向去引导，让面试官跟着你的思路走，不要让面试官随意问，如果面试官的问题是你擅长的点，一定要抓住机会，体现出你的技术深度。

（3）实质：内容听过，自我肯定；没听过，学习过程。

## 3、面试技巧

### 3.1、六个常见问题

1）你的优点是什么？

大胆的说出自己各个方面的优势和特长

2）你的缺点是什么？

不要谈为自己减分的问题；用“缺点”衬托自己的优点

3）你的离职原因是什么？

* + 不说前东家坏话，哪怕被伤过
  + 合情合理合法
  + 不要说超过1个以上的原因

4）您对薪资的期望是多少？

* + 非终面不深谈薪资
  + 只说区间，不说具体数字
  + 底线是不低于当前薪资
  + 非要具体数字，区间取中间值，或者当前薪资的+20%

5）您还有什么想问的问题？

* + 这是体现个人眼界和层次的问题
  + 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
  + 标准答案：

公司希望我入职后的3-6个月内，给公司解决什么样的问题

公司（或者对这个部门）未来的战略规划是什么样子的？

以你现在对我的了解，您觉得我需要多长时间融入公司？

6）您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前

### 3.2、两个注意事项

1）职业化的语言

2）职业化的形象

### 3.3、自我介绍

1）个人基本信息

2）工作履历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因

3）深度沟通（也叫压力面试）

刨根问底下沉式追问（注意是下沉式，而不是发散式的）

基本技巧：往自己熟悉的方向说

# 第二章 数据结构、算法、设计模式与手写代码

## 1、怎么理解时间复杂度和空间复杂度？

时间复杂度和空间复杂度一般是针对算法而言，是衡量一个算法是否高效的重要标准。先纠正一个误区，时间复杂度并不是算法执行的时间，再纠正一个误区，算法不单单指冒泡排序之类的，一个循环甚至是一个判断都可以称之为算法。其实理解起来并不冲突，八大排序甚至更多的算法本质上也是通过各种循环判断来实现的。

**时间复杂度：**指算法语句的执行次数。O(1),O(n),O(logn),O(n2)

**空间复杂度：**就是一个算法在运行过程中临时占用的存储空间大小，换句话说就是被创建次数最多的变量，它被创建了多少次，那么这个算法的空间复杂度就是多少。有个规律，如果算法语句中就有创建对象，那么这个算法的时间复杂度和空间复杂度一般一致，很好理解，算法语句被执行了多少次就创建了多少对象。

## 2、数组和链表结构简单对比？

**数组：**相同数据类型的元素按一定顺序排列的集合，就是把有限个类型相同的变量用一个名字命名，然后用编号区分他们的变量的集合，这个名字称为数组名，编号称为下标

**数组的特性：**

1.数组必须先定义固定长度，不能适应数据动态增减

2.当数据增加时，可能超出原先定义的元素个数，当数据减少时，造成内存浪费

3.数组查询比较方便，根据下标就可以直接找到元素，时间复杂度O(1)；增加和删除比较复杂，需要移动操作数所在位置后的所有数据，时间复杂度为O(N)

**链表：**是一种物理存储单元上非连续，非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

**链表的特性：**

1.链表动态进行存储分配，可适应数据动态增减

2.插入、删除数据比较方便,时间复杂度O(1)；查询必须从头开始找起，十分麻烦，时间复杂度O(N)

常见的链表:

1.单链表：通常链表每一个元素都要保存一个指向下一个元素的指针

2.双链表：每个元素既要保存到下一个元素的指针，还要保存一个上一个元素的指针

3.循环链表：在最后一个元素中下一个元素指针指向首元素

链表和数组都是在堆里分配内存

**应用：**

如果需要快速访问数据，很少或不插入和删除元素，就应该用数组；相反， 如果需要经常插入和删除元素就需要用链表数据结构了

## 3、怎么遍历一个树？

四种遍历概念

先序遍历：先访问根节点，再访问左子树，最后访问右子树。

后序遍历：先左子树，再右子树，最后根节点。

中序遍历：先左子树，再根节点，最后右子树。

层序遍历：每一层从左到右访问每一个节点。

每一个子树遍历时依然按照此时的遍历顺序。可以采用递归实现遍历。

## 4、冒泡排序（Bubble Sort）

**算法描述：**

* 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
* 针对所有的元素重复以上的步骤，除了最后一个；
* 重复步骤1~3，直到排序完成。



如果两个元素相等，不会再交换位置，所以冒泡排序是一种稳定排序算法。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \* 冒泡排序
7. \*/
8. **public** **class** BubbleSort {
10. /\*\*
11. \* @param data 被排序的数组
12. \*/
13. **public** **static** **void** bubbleSort(**int**[] data) {
15. **int** arrayLength = data.length;
17. **for** (**int** i = 1; i < arrayLength; i++) {//第i次排序
19. **for** (**int** j = 0; j < arrayLength - i; j++) {//从索引为j的数开始
20. **if** (data[j] > data[j + 1]) { //相邻元素两两对比
21. **int** temp = data[j + 1];  // 元素交换
22. data[j + 1] = data[j];
23. data[j] = temp;
24. }
25. }
27. System.out.println("第" + i + "次排序：\n" + java.util.Arrays.toString(data));
28. }
29. }
31. **public** **static** **void** main(String[] args) {
33. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
35. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
37. bubbleSort(data);
39. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
40. }
41. }

## 5、快速排序（Quick Sort）

**算法描述：**

使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

* 从数列中挑出一个元素，称为 “基准”（pivot）；
* 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
* 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。



key值的选取可以有多种形式，例如中间数或者随机数，分别会对算法的复杂度产生不同的影响。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \* 快速排序
7. \*/
8. **public** **class** QuickSort {
10. **public** **static** **void** quickSort(**int**[] data, **int** low, **int** high) {
11. **int** i, j, temp, t;
12. **if** (low > high) {
13. **return**;
14. }
15. i = low;
16. j = high;
17. //temp就是基准位
18. temp = data[low];
19. System.out.println("基准位：" + temp);
21. **while** (i < j) {
22. //先看右边，依次往左递减
23. **while** (temp <= data[j] && i < j) {
24. j--;
25. }
26. //再看左边，依次往右递增
27. **while** (temp >= data[i] && i < j) {
28. i++;
29. }
30. //如果满足条件则交换
31. **if** (i < j) {
32. System.out.println("交换：" + data[i] + "和" + data[j]);
33. t = data[j];
34. data[j] = data[i];
35. data[i] = t;
36. System.out.println(java.util.Arrays.toString(data));
38. }
39. }
40. //最后将基准位与i和j相等位置的数字交换
41. System.out.println("基准位" + temp + "和i、j相遇的位置" + data[i] + "交换");
42. data[low] = data[i];
43. data[i] = temp;
44. System.out.println(java.util.Arrays.toString(data));
46. //递归调用左半数组
47. quickSort(data, low, j - 1);
48. //递归调用右半数组
49. quickSort(data, j + 1, high);
50. }

53. **public** **static** **void** main(String[] args) {
55. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
57. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
59. quickSort(data, 0, data.length - 1);
61. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
62. }
63. }

快速排序详细参考：

<https://blog.csdn.net/shujuelin/article/details/82423852>

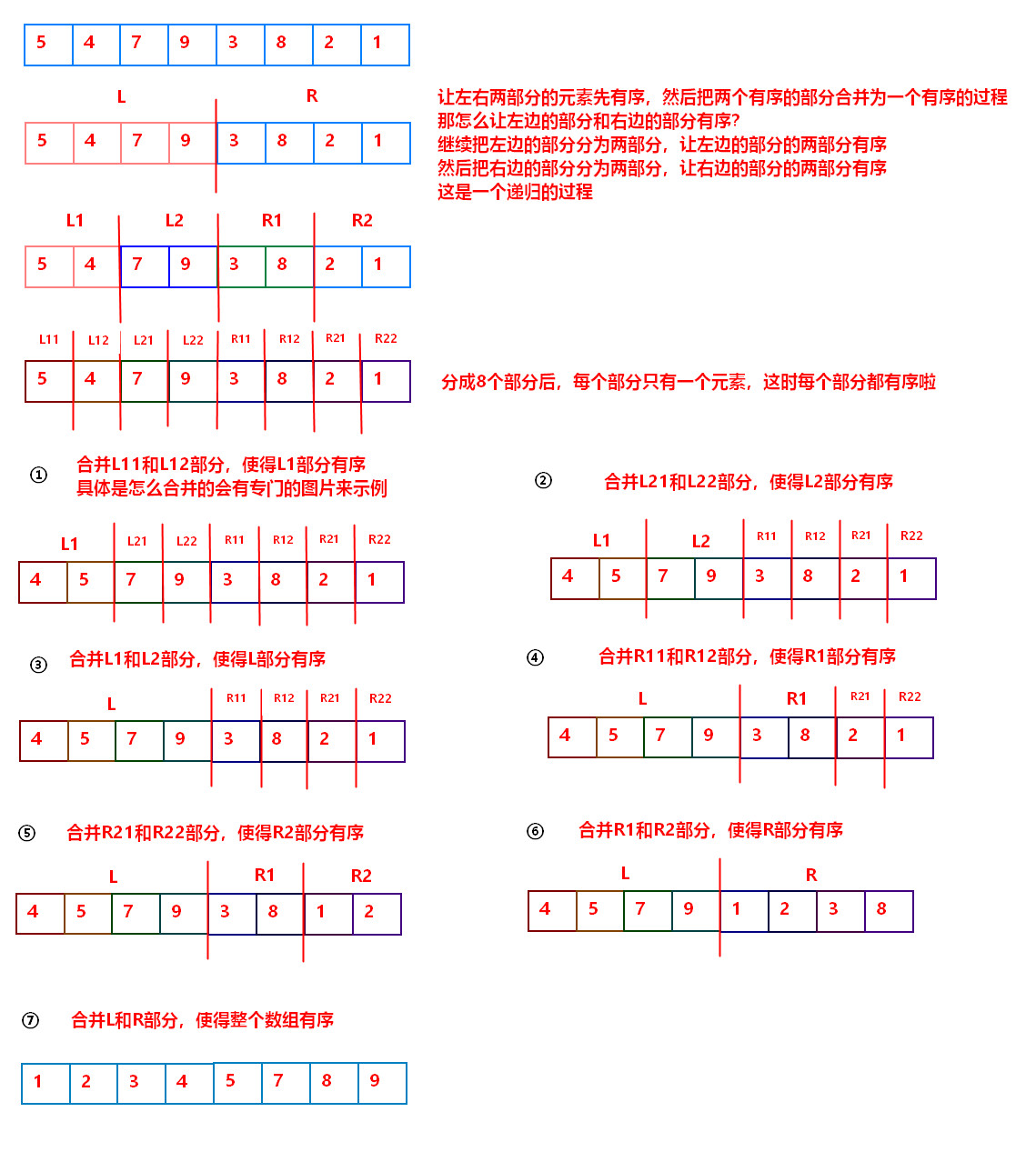
## 5、归并排序（Merge Sort）

**算法描述：**

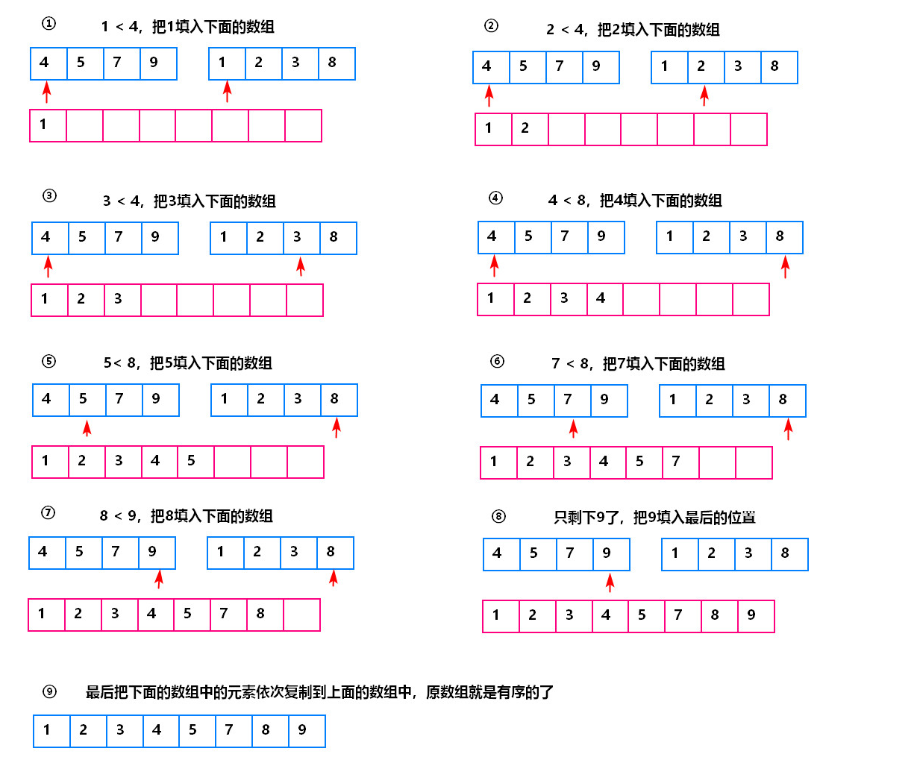
* 把长度为n的输入序列分成两个长度为n/2的子序列；
* 对这两个子序列分别采用归并排序；
* 将两个排序好的子序列合并成一个最终的排序序列。



（1）归并排序的流程



（2）合并两个有序数组的流程



**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*/
7. **public** **class** MergeSort {
9. **public** **static** **void** mergeSort(**int**[] data) {
10. sort(data, 0, data.length - 1);
11. }
13. **public** **static** **void** sort(**int**[] arr, **int** l, **int** r) {
14. **if**(l == r) {
15. **return**;
16. }
17. **int** mid = l + ((r - l) >> 1);
18. sort(arr, l, mid);
19. sort(arr, mid + 1, r);
20. merge(arr, l, mid, r);
21. }
23. **public** **static** **void** merge(**int**[] arr, **int** l, **int** mid, **int** r) {
24. **int**[] temp = **new** **int**[r - l + 1];
25. **int** i = 0;
26. **int** p1 = l;
27. **int** p2 = mid + 1;
28. // 比较左右两部分的元素，哪个小，把那个元素填入temp中
29. **while**(p1 <= mid && p2 <= r) {
30. temp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];
31. }
32. // 上面的循环退出后，把剩余的元素依次填入到temp中
33. // 以下两个while只有一个会执行
34. **while**(p1 <= mid) {
35. temp[i++] = arr[p1++];
36. }
37. **while**(p2 <= r) {
38. temp[i++] = arr[p2++];
39. }
40. // 把最终的排序的结果复制给原数组
41. **for**(i = 0; i < temp.length; i++) {
42. arr[l + i] = temp[i];
43. }
44. }
46. **public** **static** **void** main(String[] args) {
48. **int**[] data = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
50. System.out.println("排序之前：\n" + java.util.Arrays.toString(data));
52. mergeSort(data);
54. System.out.println("排序之后：\n" + java.util.Arrays.toString(data));
55. }
56. }

## 5、二分查找（Binary Search）

**算法描述：**

* 二分查找也称折半查找，它是一种效率较高的查找方法，要求列表中的元素首先要进行有序排列。
* 首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；
* 否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。
* 重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*/
7. **public** **class** BinarySearch {

10. /\*\*
11. \* 二分查找 时间复杂度O(log2n);空间复杂度O(1)
12. \*
13. \* @param arr     被查找的数组
14. \* @param left
15. \* @param right
16. \* @param findVal
17. \* @return 返回元素的索引
18. \*/
19. **public** **static** **int** binarySearch(**int**[] arr, **int** left, **int** right, **int** findVal) {
21. **if** (left > right) {//递归退出条件，找不到，返回-1
22. **return** -1;
23. }
25. **int** midIndex = (left + right) / 2;
27. **if** (findVal < arr[midIndex]) {//向左递归查找
28. **return** binarySearch(arr, left, midIndex, findVal);
29. } **else** **if** (findVal > arr[midIndex]) {//向右递归查找
30. **return** binarySearch(arr, midIndex, right, findVal);
31. } **else** {
32. **return** midIndex;
33. }
34. }
36. **public** **static** **void** main(String[] args){
38. //注意：需要对已排序的数组进行二分查找
39. **int**[] data = {-49, -30, -16, 9, 21, 21, 23, 30, 30};
40. **int** i = binarySearch(data, 0, data.length, 21);
41. System.out.println(i);
42. }
43. }

**拓展需求：**

当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到。

**代码实现：**

1. **package** com.atguigu.interview.chapter02;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
6. /\*\*
7. \* @author atguigu
8. \* @since 2019/7/22
9. \*/
10. **public** **class** BinarySearch2 {
12. /\*\*
13. \* {1, 8, 10, 89, 1000, 1000, 1234}
14. \* 一个有序数组中，有多个相同的数值，如何将所有的数值都查找到，比如这里的 1000.
15. \* 分析：
16. \* 1. 返回的结果是一个列表 list
17. \* 2. 在找到结果时，向左边扫描，向右边扫描 [条件]
18. \* 3. 找到结果后，就加入到ArrayBuffer
19. \*
20. \* @return
21. \*/
22. **public** **static** List<Integer> binarySearch2(**int**[] arr, **int** left, **int** right, **int** findVal) {
24. //找不到条件?
25. List<Integer> list = **new** ArrayList<>();
27. **if** (left > right) {//递归退出条件，找不到，返回-1
28. **return** list;
29. }
31. **int** midIndex = (left + right) / 2;
32. **int** midVal = arr[midIndex];
33. **if** (findVal < midVal) {//向左递归查找
34. **return** binarySearch2(arr, left, midIndex - 1, findVal);
35. } **else** **if** (findVal > midVal) { //向右递归查找
36. **return** binarySearch2(arr, midIndex + 1, right, findVal);
37. } **else** {
38. System.out.println("midIndex=" + midIndex);
40. //向左边扫描
41. **int** temp = midIndex - 1;
42. **while** (**true**) {
43. **if** (temp < 0 || arr[temp] != findVal) {
44. **break**;
45. }
46. **if** (arr[temp] == findVal) {
47. list.add(temp);
48. }
49. temp -= 1;
50. }
52. //将中间这个索引加入
53. list.add(midIndex);
55. //向右边扫描
56. temp = midIndex + 1;
57. **while** (**true**) {
58. **if** (temp > arr.length - 1 || arr[temp] != findVal) {
59. **break**;
60. }
61. **if** (arr[temp] == findVal) {
62. list.add(temp);
63. }
64. temp += 1;
65. }
66. **return** list;
67. }
68. }
70. **public** **static** **void** main(String[] args){
72. //注意：需要对已排序的数组进行二分查找
73. **int**[] data = {1, 8, 10, 89, 1000, 1000, 1234};
74. List<Integer> list = binarySearch2(data, 0, data.length, 1000);
75. System.out.println(list);
76. }
77. }

## 6、你所知道的设计模式有哪些？

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 7、单例模式（Binary Search）

### 7.1、单例模式定义

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态。

### 7.2、单例模式的特点

* 单例类只能有一个实例。
* 单例类必须自己创建自己的唯一实例。
* 单例类必须给所有其他对象提供这一实例。

单例模式保证了全局对象的唯一性，比如系统启动读取配置文件就需要单例保证配置的一致性。

### 7.3、单例的四大原则

* 构造私有
* 以静态方法或者枚举返回实例
* 确保实例只有一个，尤其是多线程环境
* 确保反序列化时不会重新构建对象
* 防止反射攻击

### 7.4、实现单例模式的方式

**（1）饿汉式（立即加载）：**

饿汉式单例在类加载初始化时就创建好一个静态的对象供外部使用，除非系统重启，这个对象不会改变，所以本身就是线程安全的。

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。（事实上，通过Java反射机制是能够实例化构造方法为private的类的，会使Java单例实现失效）

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*
7. \* 饿汉式（立即加载）
8. \*/
9. **public** **class** Singleton1 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton1() {
15. System.out.println("构造函数Singleton1");
16. }
18. /\*\*
19. \* 初始值为实例对象
20. \*/
21. **private** **static** Singleton1 single = **new** Singleton1();
23. /\*\*
24. \* 静态工厂方法
25. \* @return 单例对象
26. \*/
27. **public** **static** Singleton1 getInstance() {
28. System.out.println("getInstance");
29. **return** single;
30. }
32. **public** **static** **void** main(String[] args){
33. System.out.println("初始化");
34. Singleton1 instance = Singleton1.getInstance();
35. }
36. }
37. **懒汉式（延迟加载）：**

该示例虽然用延迟加载方式实现了懒汉式单例，但在多线程环境下会产生多个Singleton对象

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*
7. \* 懒汉式（延迟加载）
8. \*/
9. **public** **class** Singleton2 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton2() {
15. System.out.println("构造函数Singleton2");
16. }
18. /\*\*
19. \* 初始值为null
20. \*/
21. **private** **static** Singleton2 single = null;
23. /\*\*
24. \* 静态工厂方法
25. \* @return 单例对象
26. \*/
27. **public** **static** Singleton2 getInstance() {
28. **if**(single == null){
29. System.out.println("getInstance");
30. single = **new** Singleton2();
31. }
32. **return** single;
33. }
35. **public** **static** **void** main(String[] args){
37. System.out.println("初始化");
38. Singleton2 instance = Singleton2.getInstance();
39. }
40. }
41. **同步锁（解决线程安全问题）：**

在方法上加synchronized同步锁或是用同步代码块对类加同步锁，此种方式虽然解决了多个实例对象问题，但是该方式运行效率却很低下，下一个线程想要获取对象，就必须等待上一个线程释放锁之后，才可以继续运行。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*
7. \* 同步锁（解决线程安全问题）
8. \*/
9. **public** **class** Singleton3 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton3() {}
16. /\*\*
17. \* 初始值为null
18. \*/
19. **private** **static** Singleton3 single = null;
21. **public** **static** synchronized Singleton3 getInstance() {
23. **if**(single == null){
24. single = **new** Singleton3();
25. }
26. **return** single;
27. }
28. }

**（4）双重检查锁（提高同步锁的效率）：**

使用双重检查锁进一步做了优化，可以避免整个方法被锁，只对需要锁的代码部分加锁，可以提高执行效率。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \* 双重检查锁（提高同步锁的效率）
7. \*/
8. **public** **class** Singleton4 {
10. /\*\*
11. \* 私有构造
12. \*/
13. **private** Singleton4() {}
15. /\*\*
16. \* 初始值为null
17. \*/
18. **private** **static** Singleton4 single = null;
20. /\*\*
21. \* 双重检查锁
22. \* @return 单例对象
23. \*/
24. **public** **static** Singleton4 getInstance() {
25. **if** (single == null) {
26. synchronized (Singleton4.**class**) {
27. **if** (single == null) {
28. single = **new** Singleton4();
29. }
30. }
31. }
32. **return** single;
33. }
34. }

**（5） 静态内部类：**

这种方式引入了一个内部静态类（static class），静态内部类只有在调用时才会加载，它保证了Singleton 实例的延迟初始化，又保证了实例的唯一性。它把singleton 的实例化操作放到一个静态内部类中，在第一次调用getInstance() 方法时，JVM才会去加载InnerObject类，同时初始化singleton 实例，所以能让getInstance() 方法线程安全。

特点是：即能延迟加载，也能保证线程安全。

静态内部类虽然保证了单例在多线程并发下的线程安全性，但是在遇到序列化对象时，默认的方式运行得到的结果就是多例的。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*
7. \* 静态内部类（延迟加载，线程安全）
8. \*/
9. **public** **class** Singleton5 {
11. /\*\*
12. \* 私有构造
13. \*/
14. **private** Singleton5() {}
16. /\*\*
17. \* 静态内部类
18. \*/
19. **private** **static** **class** InnerObject{
20. **private** **static** Singleton5 single = **new** Singleton5();
21. }
23. **public** **static** Singleton5 getInstance() {
24. **return** InnerObject.single;
25. }
26. }

**（6）内部枚举类实现（防止反射攻击和反序列化）：**

事实上，通过Java反射机制是能够实例化构造方法为private的类的。这也就是我们现在需要引入的枚举单例模式。

1. package com.atguigu.interview.chapter02;
3. /\*\*
4. \* @author atguigu
5. \* @since 2019/7/22
6. \*/
7. **public** **class** SingletonFactory {
9. /\*\*
10. \* 内部枚举类
11. \*/
12. **private** **enum** EnumSingleton{
13. Singleton;
14. **private** Singleton6 singleton;
16. //枚举类的构造方法在类加载是被实例化
17. **private** EnumSingleton(){
18. singleton = **new** Singleton6();
19. }
20. **public** Singleton6 getInstance(){
21. **return** singleton;
22. }
23. }
25. **public** **static** Singleton6 getInstance() {
26. **return** EnumSingleton.Singleton.getInstance();
27. }
28. }
30. **class** Singleton6 {
31. **public** Singleton6(){}
32. }

## 8、工厂设计模式（Factory）

### 8.1、什么是工厂设计模式？

工厂设计模式，顾名思义，就是用来生产对象的，在java中，万物皆对象，这些对象都需要创建，如果创建的时候直接new该对象，就会对该对象耦合严重，假如我们要更换对象，所有new对象的地方都需要修改一遍，这显然违背了软件设计的开闭原则，如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：解耦

### 8.2、简单工厂（Simple Factory）

**定义：**

一个工厂方法，依据传入的参数，生成对应的产品对象；  
**角色：**  
1、抽象产品  
2、具体产品  
3、具体工厂  
4、产品使用者  
**使用说明：**

先将产品类抽象出来，比如，苹果和梨都属于水果，抽象出来一个水果类Fruit，苹果和梨就是具体的产品类，然后创建一个水果工厂，分别用来创建苹果和梨。代码如下：

**水果接口：**

1. **public** interface Fruit {
2. **void** whatIm();
3. }

**苹果类：**

1. **public** **class** Apple **implements** Fruit {
2. @Override
3. **public** **void** whatIm() {
4. System.out.println("苹果");
5. }
6. }

**梨类：**

1. **public** **class** Pear **implements** Fruit {
2. @Override
3. **public** **void** whatIm() {
4. System.out.println("梨");
5. }
6. }

**水果工厂：**

1. **public** **class** FruitFactory {
3. **public** Fruit createFruit(String type) {
5. **if** (type.equals("apple")) {//生产苹果
6. **return** **new** Apple();
7. } **else** **if** (type.equals("pear")) {//生产梨
8. **return** **new** Pear();
9. }
11. **return** null;
12. }
13. }

**使用工厂生产产品：**

1. **public** **class** FruitApp {
3. **public** **static** **void** main(String[] args) {
4. FruitFactory mFactory = **new** FruitFactory();
5. Apple apple = (Apple) mFactory.createFruit("apple");//获得苹果
6. Pear pear = (Pear) mFactory.createFruit("pear");//获得梨
7. apple.whatIm();
8. pear.whatIm();
9. }
10. }

以上的这种方式，每当添加一种水果，就必然要修改工厂类，违反了开闭原则；

所以简单工厂只适合于产品对象较少，且产品固定的需求，对于产品变化无常的需求来说显然不合适。

### 8.3、工厂方法（Factory Method）

**定义：**

将工厂提取成一个接口或抽象类，具体生产什么产品由子类决定；  
**角色：**  
1、抽象产品  
2、具体产品  
3、抽象工厂  
4、具体工厂  
**使用说明：**

和上例中一样，产品类抽象出来，这次我们把工厂类也抽象出来，生产什么样的产品由子类来决定。代码如下：  
**水果接口、苹果类和梨类：**

代码和上例一样

**抽象工厂接口：**

1. **public** interface FruitFactory {
2. Fruit createFruit();//生产水果
3. }

**苹果工厂：**

1. **public** **class** AppleFactory implements FruitFactory {
2. @Override
3. **public** Apple createFruit() {
4. **return** **new** Apple();
5. }
6. }

**梨工厂：**

1. **public** **class** PearFactory implements FruitFactory {
2. @Override
3. **public** Pear createFruit() {
4. **return** **new** Pear();
5. }
6. }

**使用工厂生产产品：**

1. **public** **class** FruitApp {
3. **public** **static** **void** main(String[] args){
4. AppleFactory appleFactory = **new** AppleFactory();
5. PearFactory pearFactory = **new** PearFactory();
6. Apple apple = appleFactory.createFruit();//获得苹果
7. Pear pear = pearFactory.createFruit();//获得梨
8. apple.whatIm();
9. pear.whatIm();
10. }
11. }

以上这种方式，虽然解耦了，也遵循了开闭原则，但是如果我需要的产品很多的话，需要创建非常多的工厂，所以这种方式的缺点也很明显。

### 8.4、抽象工厂（Abstract Factory）

**定义：**

为创建一组相关或者是相互依赖的对象提供的一个接口，而不需要指定它们的具体类。  
**角色：**

1. 抽象产品  
   2、具体产品  
   3、抽象工厂  
   4、具体工厂

**使用说明：**

抽象工厂和工厂方法的模式基本一样，区别在于，工厂方法是生产一个具体的产品，而抽象工厂可以用来生产一组相同，有相对关系的产品；重点在于一组，一批，一系列；举个例子，假如生产小米手机，小米手机有很多系列，小米note、红米note等；假如小米note生产需要的配件有825的处理器，6英寸屏幕，而红米只需要650的处理器和5寸的屏幕就可以了。用抽象工厂来实现：

**cpu接口和实现类：**

1. **public** interface Cpu {
2. **void** run();
4. **class** Cpu650 implements Cpu {
5. @Override
6. **public** **void** run() {
7. System.out.println("650 也厉害");
8. }
9. }
11. **class** Cpu825 implements Cpu {
12. @Override
13. **public** **void** run() {
14. System.out.println("825 更强劲");
15. }
16. }
17. }

**屏幕接口和实现类：**

1. **public** interface Screen {
3. **void** size();
5. **class** Screen5 implements Screen {
7. @Override
8. **public** **void** size() {
9. System.out.println("" +
10. "5寸");
11. }
12. }
14. **class** Screen6 implements Screen {
16. @Override
17. **public** **void** size() {
18. System.out.println("6寸");
19. }
20. }
21. }

**抽象工厂接口：**

1. **public** interface PhoneFactory {
3. Cpu getCpu();//使用的cpu
5. Screen getScreen();//使用的屏幕
6. }

**小米手机工厂：**

1. **public** **class** XiaoMiFactory implements PhoneFactory {
2. @Override
3. **public** Cpu.Cpu825 getCpu() {
4. **return** **new** Cpu.Cpu825();//高性能处理器
5. }
7. @Override
8. **public** Screen.Screen6 getScreen() {
9. **return** **new** Screen.Screen6();//6寸大屏
10. }
11. }

**红米手机工厂：**

1. **public** **class** HongMiFactory implements PhoneFactory {
3. @Override
4. **public** Cpu.Cpu650 getCpu() {
5. **return** **new** Cpu.Cpu650();//高效处理器
6. }
8. @Override
9. **public** Screen.Screen5 getScreen() {
10. **return** **new** Screen.Screen5();//小屏手机
11. }
12. }

**使用工厂生产产品：**

1. **public** **class** PhoneApp {
2. **public** **static** **void** main(String[] args){
3. HongMiFactory hongMiFactory = **new** HongMiFactory();
4. XiaoMiFactory xiaoMiFactory = **new** XiaoMiFactory();
5. Cpu.Cpu650 cpu650 = hongMiFactory.getCpu();
6. Cpu.Cpu825 cpu825 = xiaoMiFactory.getCpu();
7. cpu650.run();
8. cpu825.run();
10. Screen.Screen5 screen5 = hongMiFactory.getScreen();
11. Screen.Screen6 screen6 = xiaoMiFactory.getScreen();
12. screen5.size();
13. screen6.size();
14. }
15. }

以上例子可以看出，抽象工厂可以解决一系列的产品生产的需求，对于大批量，多系列的产品，用抽象工厂可以更好的管理和扩展。

### 8.5、三种工厂方式总结

1、对于简单工厂和工厂方法来说，两者的使用方式实际上是一样的，如果对于产品的分类和名称是确定的，数量是相对固定的，推荐使用简单工厂模式；

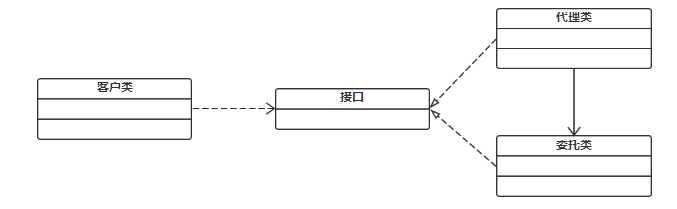
2、抽象工厂用来解决相对复杂的问题，适用于一系列、大批量的对象生产。

## 9、代理模式（Proxy）

### 9.1、什么是代理模式？

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了。用图表示如下：



### 9.2、为什么要用代理模式？

**中介隔离作用：**

在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

**开闭原则，增加功能：**

代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要修改已经封装好的委托类。

### 9.3、有哪几种代理模式？

我们有多种不同的方式来实现代理。

如果按照代理创建的时期来进行分类的话，可以分为两种：静态代理、动态代理。

* 静态代理是由程序员创建或特定工具自动生成源代码，再对其编译。在程序员运行之前，代理类.class文件就已经被创建了。
* 动态代理是在程序运行时通过反射机制动态创建的。

### 9.4、静态代理（Static Proxy）

**第一步：创建服务类接口**

1. **public** interface BuyHouse {
2. **void** buyHouse();
3. }

**第二步：实现服务接口**

1. **public** **class** BuyHouseImpl implements BuyHouse {
3. @Override
4. **public** **void** buyHouse() {
5. System.out.println("我要买房");
6. }
7. }

**第三步：创建代理类**

1. **public** **class** BuyHouseProxy implements BuyHouse {
3. **private** BuyHouse buyHouse;
5. **public** BuyHouseProxy(final BuyHouse buyHouse) {
6. **this**.buyHouse = buyHouse;
7. }
9. @Override
10. **public** **void** buyHouse() {
11. System.out.println("买房前准备");
12. buyHouse.buyHouse();
13. System.out.println("买房后装修");
15. }
16. }

**第四步：编写测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
4. BuyHouse buyHouse = **new** BuyHouseImpl();
5. BuyHouseProxy buyHouseProxy = **new** BuyHouseProxy(buyHouse);
6. buyHouseProxy.buyHouse();
7. }
8. }

**静态代理总结：**

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点：我们得为每一个服务创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

### 9.5 、JDK动态代理（Dynamic Proxy）

在动态代理中我们不再需要再手动的创建代理类，我们只需要编写一个动态处理器就可以了。真正的代理对象由JDK在运行时为我们动态的来创建。

**第一步：创建服务类接口**

代码和上例一样

**第二步：实现服务接口**

代码和上例一样

**第三步：编写动态处理器**

1. **public** **class** DynamicProxyHandler implements InvocationHandler {
3. **private** Object object;
5. **public** DynamicProxyHandler(final Object object) {
6. **this**.object = object;
7. }
9. @Override
10. **public** Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
11. System.out.println("买房前准备");
12. Object result = method.invoke(object, args);
13. System.out.println("买房后装修");
14. **return** result;
15. }
16. }

**第四步：编写测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
4. BuyHouse buyHouse = **new** BuyHouseImpl();
5. BuyHouse proxyBuyHouse = (BuyHouse) Proxy.newProxyInstance(
6. BuyHouse.**class**.getClassLoader(),
7. **new** Class[]{BuyHouse.**class**},
8. **new** DynamicProxyHandler(buyHouse));
9. proxyBuyHouse.buyHouse();
10. }
11. }

 Proxy是所有动态生成的代理的共同的父类，这个类有一个静态方法Proxy.newProxyInstance()，接收三个参数：

* ClassLoader loader：指定当前目标对象使用的类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces：指定目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler：指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

**JDK动态代理总结：**

优点：相对于静态代理，动态代理大大减少了开发任务，同时减少了对业务接口的依赖，降低了耦合度。

缺点：Proxy是所有动态生成的代理的共同的父类，因此服务类必须是接口的形式，不能是普通类的形式，因为Java无法实现多继承。

### 9.6 、CGLib动态代理（CGLib Proxy）

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

**Cglib子类代理实现方法：**

（1）引入cglib的jar文件，asm的jar文件

（2）代理的类不能为final

（3）目标业务对象的方法如果为final/static，那么就不会被拦截，即不会执行目标对象额外的业务方法

**第一步：创建服务类**

1. **public** **class** BuyHouse2 {
3. **public** **void** buyHouse() {
4. System.out.println("我要买房");
5. }
6. }

**第二步：创建CGLIB代理类**

1. **public** **class** CglibProxy implements MethodInterceptor {
3. **private** Object target;
5. **public** CglibProxy(Object target) {
6. **this**.target = target;
7. }
9. /\*\*
10. \*  给目标对象创建一个代理对象
11. \* @return 代理对象
12. \*/
13. **public** Object getProxyInstance() {
14. //1.工具类
15. Enhancer enhancer = **new** Enhancer();
16. //2.设置父类
17. enhancer.setSuperclass(target.getClass());
18. //3.设置回调函数
19. enhancer.setCallback(**this**);
20. //4.创建子类(代理对象)
21. **return** enhancer.create();
23. }
25. **public** Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
26. System.out.println("买房前准备");
27. //执行目标对象的方法
28. Object result = method.invoke(target, args);
29. System.out.println("买房后装修");
30. **return** result;
31. }
32. }

**第三步：创建测试类**

1. **public** **class** HouseApp {
3. **public** **static** **void** main(String[] args) {
5. BuyHouse2 target = **new** BuyHouse2();
6. CglibProxy cglibProxy = **new** CglibProxy(target);
7. BuyHouse2 buyHouseCglibProxy = (BuyHouse2) cglibProxy.getProxyInstance();
8. buyHouseCglibProxy.buyHouse();
9. }
10. }

**CGLib代理总结：**

CGLib创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

### 9.7、简述动态代理的原理， 常用的动态代理的实现方式

动态代理的原理: **使用一个代理将对象包装起来**，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。

代理对象决定是否以及何时将方法调 用转到原始对象上

动态代理的方式

基于接口实现动态代理： JDK动态代理

基于继承实现动态代理： Cglib、Javassist动态代理

# 第三章 Java基础篇

## 1、你是怎样理解面向对象的

面向对象是利于语言对现实事物进行抽象。面向对象具有以下四大特征：

**（1）继承：**继承是从已有类得到继承信息创建新类的过程

**（2）封装：**通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

**（3）多态性：**多态性是指允许不同子类型的对象对同一消息作出不同的响应。

**（4）抽象：**抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。

## 2、你是怎样理解多态的？什么地方用过？

同一个行为具有多个不同表现形式或形态的能力。

父类引用指向子类对象，例如 List<String> list = new ArrayList<String>();就是典型的一种多态的体现形式。

## 3、举例说明封装和继承是怎么回事?

**封装**是指一种将抽象性函式接口的实现细节部分包装、隐藏起来的方法。

例如：实体类的封装，下面代码中，将 name 和 age 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问。

public class Person{

private String name;

private int age; ​

public int getAge(){ return age; } ​

public String getName(){ return name; } ​

public void setAge(int age){ this.age = age; } ​

public void setName(String name){ this.name = name; }

}

**继承**就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

## 4、char型变量能不能储存一个中文汉字，为什么？

char型变量是用来存储Unicode编码的字符的，unicode编码字符集中包含了汉字，所以，char型变量中当然可以存储汉字。不过，如果某个特殊的汉字没有被包含在unicode编码字符集中，那么，这个char型变量中就不能存储这个特殊汉字。说明：unicode编码占用两个字节，所以，char类型的变量也是占用两个字节。

## 5、什么是自动拆装箱？ int和Integer有什么区别?以及以下程序运行结果。

基本数据类型，如int,float,double,boolean,char,byte,不具备对象的特bai征，不能调用方法du。

**装箱：**将基本类型转换成包装类对象

**拆箱：**将包装类对象转换成基本类型的值

java为什么要引入自动装箱和拆箱的功能？主要是用于java集合中，List<Inteter> list=new ArrayList<Integer>();

list集合如果要放整数的话，只能放对象，不能放基本类型，因此需要将整数自动装箱成对象。

**实现原理：**javac编译器的语法糖

**区别：**

（1）Integer是int的包装类，int则是java的一种基本数据类型

（2）Integer变量必须实例化后才能使用，而int变量不需要

（3）Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值

（4）Integer的默认值是null，int的默认值是0

1. package com.atguigu.interview.chapter03;
2. **public** **class** Test01 {
3. **public** **static** **void** main(String[] args){
4. Integer a = 127;
5. Integer b = 127;
6. Integer c = 128;
7. Integer d = 128;
8. System.out.println(a==b); //true
9. System.out.println(c==d); //false
10. }
11. }

## 6、==和Equals区别

（1） ==

如果比较的是基本数据类型，那么比较的是变量的值

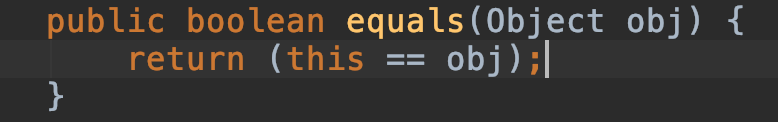
如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

1. equals

如果没重写equals方法比较的是两个对象的地址值

如果重写了equals方法后我们往往比较的是对象中的属性的内容

equals方法是从Object类中继承的，默认的实现就是使用==



## 7、String能被继承吗？为什么用final修饰？

不能被继承，因为String类有final修饰符，而final修饰的类是不能被继承的。

* String 类是最常用的类之一，为了效率，禁止被继承和重写。
* 为了安全。String 类中有很多调用底层的本地方法，调用了操作系统的 API，如果方法可以重写，可能被植入恶意代码，破坏程序。Java 的安全性也体现在这里。

## 8、String buffer和String builder区别

（1）StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，

（2）只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。

（3）在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低

## 9、Final、Finally、Finalize

**final：**修饰符（关键字）有三种用法：修饰类、变量和方法。修饰类时，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。修饰变量时，该变量使用中不被改变，必须在声明时给定初值，在引用中只能读取不可修改，即为常量。修饰方法时，也同样只能使用，不能在子类中被重写。

**finally：**通常放在try…catch的后面构造最终执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

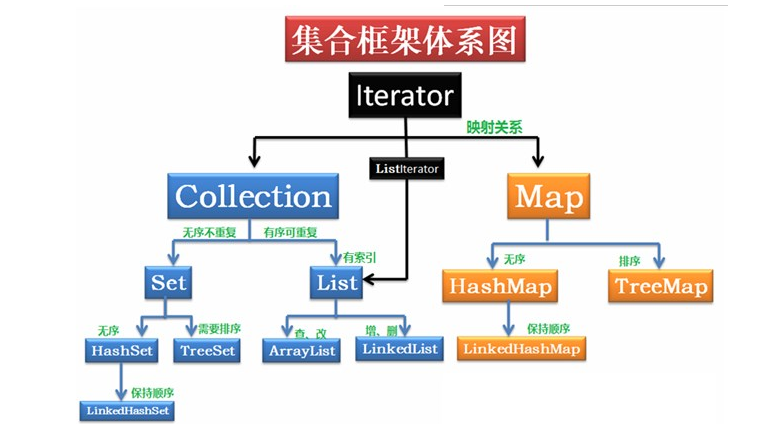
**finalize：**Object类中定义的方法，Java中允许使用finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize() 方法可以整理系统资源或者执行其他清理工作。

## 10、Object中有哪些方法

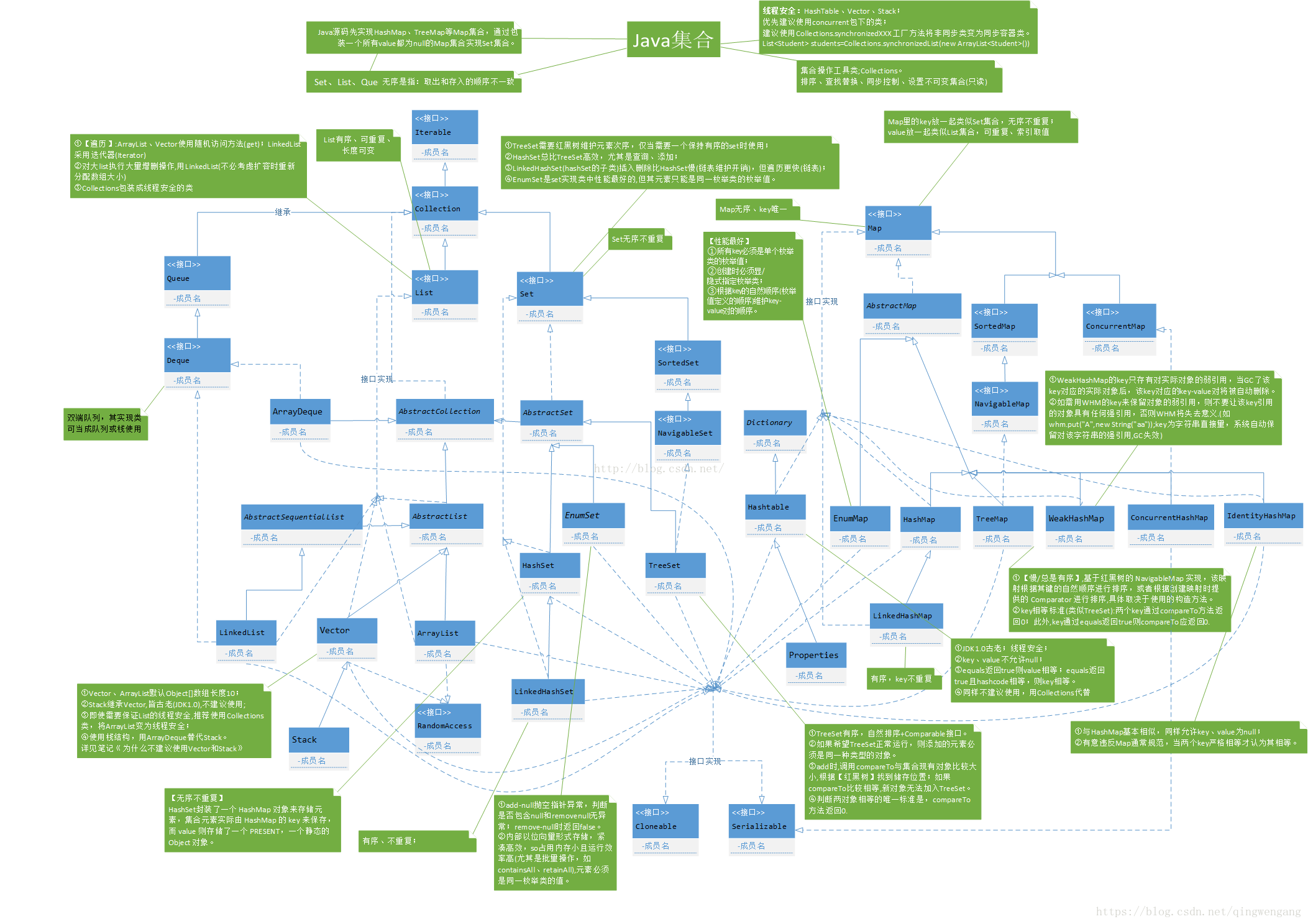
（1）protected Object clone()--->创建并返回此对象的一个副本。   
（2）boolean equals(Object obj)--->指示某个其他对象是否与此对象“相等”。   
（3）protected void finalize()--->当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。   
（4）Class<? extendsObject> getClass()--->返回一个对象的运行时类。   
（5）int hashCode()--->返回该对象的哈希码值。   
（6）void notify()--->唤醒在此对象监视器上等待的单个线程。   
（7）void notifyAll()--->唤醒在此对象监视器上等待的所有线程。   
（8）String toString()--->返回该对象的字符串表示。   
（9）void wait()--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。   
 void wait(long timeout)--->导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll()方法，或者超过指定的时间量。   
 void wait(long timeout, int nanos)--->导致当前的线程等待，直到其他线程调用此对象的 notify()

## 11、说一下集合体系？ArrarList和LinkedList区别

**简单体系：**



**复杂体系：**



**ArrarList和LinkedList区别：**

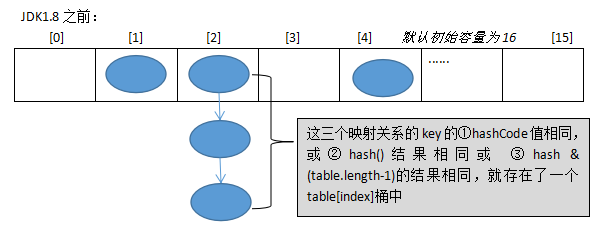
（1）ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

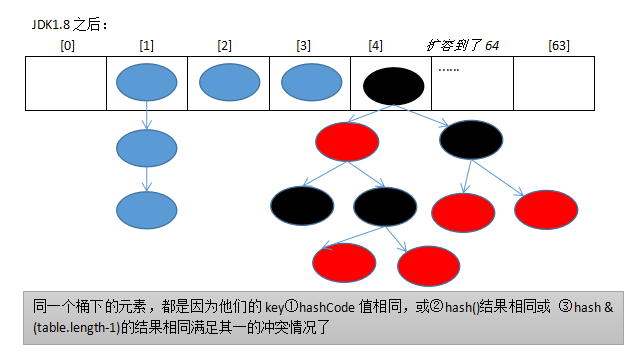
（2）对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

（3）对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。 这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

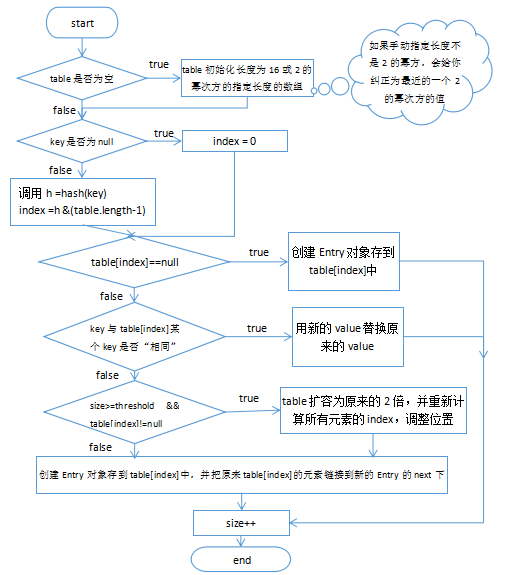
## 12 、HashMap底层源码，数据结构

HashMap的底层结构在jdk1.7中由数组+链表实现，在jdk1.8中由数组+链表+红黑树实现，以数组+链表的结构为例。

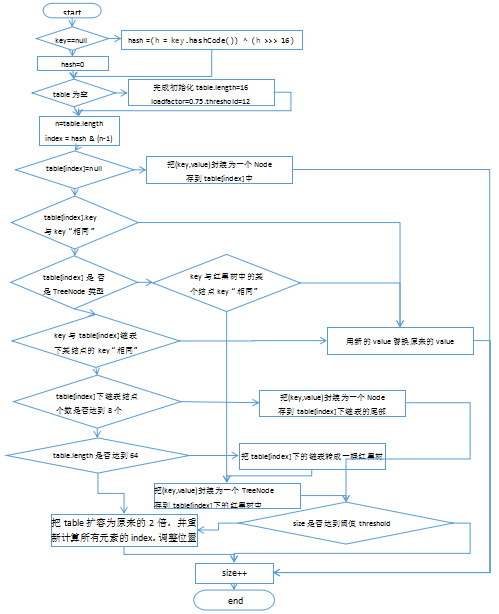




**JDK1.8之前Put方法：**



**JDK1.8之后Put方法：**



**注意：可以结合百度hashmap源码解析进行更深入的了解。**

## 13 、HashMap和HashTable区别

**（1）线程安全性不同**

HashMap是线程不安全的，HashTable是线程安全的，其中的方法是Synchronize的，在多线程并发的情况下，可以直接使用HashTabl，但是使用HashMap时必须自己增加同步处理。

**（2）是否提供contains方法**

HashMap只有containsValue和containsKey方法；HashTable有contains、containsKey和containsValue三个方法，其中contains和containsValue方法功能相同。

**（3）key和value是否允许null值**

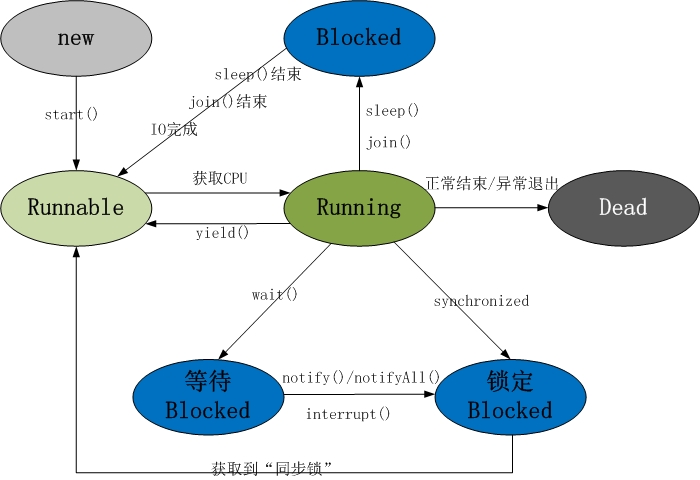
Hashtable中，key和value都不允许出现null值。HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

**（4）数组初始化和扩容机制**

HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

## 14、线程的状态转换有什么？（生命周期）



1. 新建状态(New) ：线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。  
   （2）就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。  
   （3）运行状态(Running)：线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。  
   （4）阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

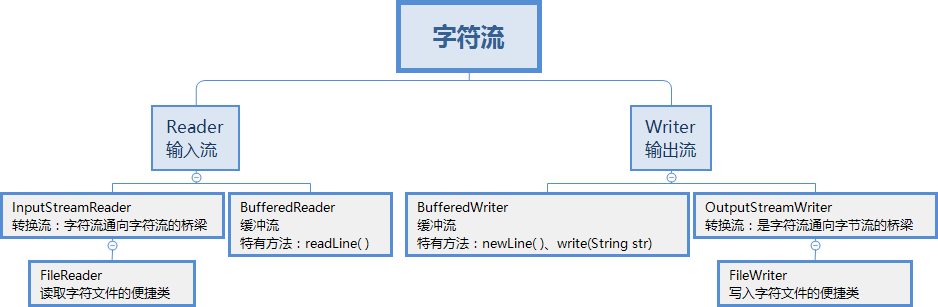
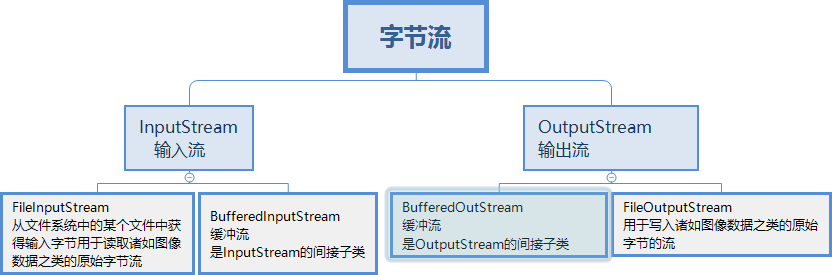
* 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。
* 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。
* 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join(）等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

（5）死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 15、什么情况下会产生死锁？

**产生死锁的原因：**  
（1） 因为系统资源不足。  
（2） 进程/线程运行推进的顺序不合适。  
（3） 资源分配不当等。  
 如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则  
就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。  
**产生死锁的四个必要条件：**（1） 互斥条件：一个资源每次只能被一个进程使用。  
（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。  
（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  
（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。  
 这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。  
**死锁的解除与预防：**  
 理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和  
解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确  
定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态  
的情况下占用资源。因此，对资源的分配要给予合理的规划。

## 16 、Java 中有几种类型的流



## 17、字节流如何转为字符流

字节输入流转字符输入流通过 InputStreamReader 实现，该类的构造函数可以传入 InputStream 对象。

字节输出流转字符输出流通过OutputStreamWriter 实现，该类的构造函数可以传入 OutputStream 对象。

## 18、请写出你最常见的5个RuntimeException

（1）java.lang.NullPointerException 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象。

（2）java.lang.ClassNotFoundException 指定的类找不到；出现原因：类的名称和路径加载错误；通常都是程序试图通过字符串来加载某个类时可能引发异常。

（3）java.lang.NumberFormatException 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符。

（4）java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。

（5）java.lang.IllegalArgumentException 方法传递参数错误。

（6）java.lang.ClassCastException 数据类型转换异常。

## 19、谈谈你对反射的理解

**（1）反射机制:**

所谓的反射机制就是java语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。

Java的反射机制的实现要借助于4个类：class，Constructor，Field，Method;

其中class代表的时类对 象，Constructor－类的构造器对象，Field－类的属性对象，Method－类的方法对象。通过这四个对象我们可以粗略的看到一个类的各个组 成部分。

**（2）Java反射的作用：**

在Java运行时环境中，对于任意一个类，可以知道这个类有哪些属性和方法。对于任意一个对象，可以调用它的任意一个方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。

**（3）Java 反射机制提供功能**

在运行时判断任意一个对象所属的类。

在运行时构造任意一个类的对象。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法

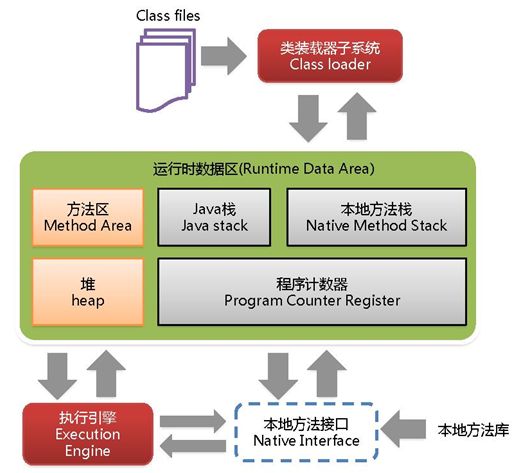
## 20、什么是 java 序列化，如何实现 java 序列化？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

序 列 化 的 实 现 ： 将 需 要 被 序 列 化 的 类 实 现 Serializable 接 口 ， 该 接 口 没 有 需 要 实 现 的 方 法 ， implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

# 第四章 Java高级篇

## 1、JVM内存分哪几个区，每个区的作用是什么?



java虚拟机主要分为以下几个区:

**（1）方法区**：

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
3. 该区域是被线程共享的。
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

**（2）虚拟机栈**:

1. 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
2. 虚拟机栈是线程私有的，它的生命周期与线程相同。
3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
4. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
5. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

**（3）本地方法栈**：  
本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

**（4）堆**：

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

**（5）程序计数器：**

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

## 2、java中垃圾收集的方法有哪些?

**1）引用计数法**   应用于：微软的COM/ActionScrip3/Python等

a) 如果对象没有被引用，就会被回收，缺点：需要维护一个引用计算器

**2）复制算法** 年轻代中使用的是Minor GC，这种GC算法采用的是复制算法(Copying)

a) 效率高，缺点：需要内存容量大，比较耗内存

b) 使用在占空间比较小、刷新次数多的新生区

**3）标记清除** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率比较低，会差生碎片。

**4）标记压缩** 老年代一般是由标记清除或者是标记清除与标记整理的混合实现

a) 效率低速度慢，需要移动对象，但不会产生碎片。

**5）标记清除压缩**标记清除-标记压缩的集合，多次GC后才Compact

a) 使用于占空间大刷新次数少的养老区，是3 4的集合体

## 3、如何判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

**（1）引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

**（2）可达性算法(引用链法)**

该算法的基本思路就是通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。

在java中可以作为GC Roots的对象有以下几种：虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈JNI引用的对象。

## 4、什么情况下会产生StackOverflowError（栈溢出）和OutOfMemoryError（堆溢出）？怎么排查？

引发 StackOverFlowError 的常见原因有以下几种：

* 无限递归循环调用（最常见）。
* 执行了大量方法，导致线程栈空间耗尽。
* 方法内声明了海量的局部变量。
* native 代码有栈上分配的逻辑，并且要求的内存还不小，比如 java.net.SocketInputStream.read0 会在栈上要求分配一个 64KB 的缓存（64位 Linux）。

引发 OutOfMemoryError的常见原因有以下几种：

* 内存bai中加载的数据量过于庞大，如一次从数据库取出过多数据。
* 集合类中有对对象的引用，使用完后未清空，dao使得JVM不能回收。
* 代码中存在死循环或循环产生过多重复的对象实体。
* 启动参数内存值设定的过小。

排查：可以通过jvisualvm进行内存快照分析，参考<https://www.cnblogs.com/boboooo/p/13164071.html>

## 5、什么是线程池，线程池有哪些？

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率

在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法。

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);

ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(4);

ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();

然后调用他们的 execute 方法即可。

**（1）newCachedThreadPool**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**（2）newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

**（3）newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**（4）newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行。例如延迟3秒执行。

这4种线程池底层 全部是ThreadPoolExecutor对象的实现，阿里规范手册中规定线程池采用ThreadPoolExecutor自定义的，实际开发也是。

## 6、ThreadPoolExecutor对象有哪些参数？都有什么作用？怎么设定核心线程数和最大线程数？拒绝策略有哪些？

**参数与作用：**共7个参数

**corePoolSize：**核心线程数，在ThreadPoolExecutor中有一个与它相关的配置：allowCoreThreadTimeOut（默认为false），当allowCoreThreadTimeOut为false时，核心线程会一直存活，哪怕是一直空闲着。而当allowCoreThreadTimeOut为true时核心线程空闲时间超过keepAliveTime时会被回收。

**maximumPoolSize：**最大线程数，线程池能容纳的最大线程数，当线程池中的线程达到最大时，此时添加任务将会采用拒绝策略，默认的拒绝策略是抛出一个运行时错误（RejectedExecutionException）。值得一提的是，当初始化时用的工作队列为LinkedBlockingDeque时，这个值将无效。

**keepAliveTime：**存活时间，当非核心空闲超过这个时间将被回收，同时空闲核心线程是否回收受allowCoreThreadTimeOut影响。

**unit：**keepAliveTime的单位。

**workQueue：**任务队列，常用有三种队列，即SynchronousQueue,LinkedBlockingDeque（无界队列）,ArrayBlockingQueue（有界队列）。

**threadFactory：**线程工厂，ThreadFactory是一个接口，用来创建worker。通过线程工厂可以对线程的一些属性进行定制。默认直接新建线程。

**RejectedExecutionHandler：**也是一个接口，只有一个方法，当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution法。

默认是抛出一个运行时异常。

**线程池大小设置：**

1. 需要分析线程池执行的任务的特性： CPU 密集型还是 IO 密集型
2. 每个任务执行的平均时长大概是多少，这个任务的执行时长可能还跟任务处理逻辑是否涉及到网络传输以及底层系统资源依赖有关系

如果是 CPU 密集型，主要是执行计算任务，响应时间很快，cpu 一直在运行，这种任务 cpu的利用率很高，那么线程数的配置应该根据 CPU 核心数来决定，CPU 核心数=最大同时执行线程数，加入 CPU 核心数为 4，那么服务器最多能同时执行 4 个线程。过多的线程会导致上下文切换反而使得效率降低。那线程池的最大线程数可以配置为 cpu 核心数+1 如果是 IO 密集型，主要是进行 IO 操作，执行 IO 操作的时间较长，这是 cpu 出于空闲状态，导致 cpu 的利用率不高，这种情况下可以增加线程池的大小。这种情况下可以结合线程的等待时长来做判断，等待时间越高，那么线程数也相对越多。一般可以配置 cpu 核心数的 2 倍。

一个公式：线程池设定最佳线程数目 = （（线程池设定的线程等待时间+线程 CPU 时间）/  
线程 CPU 时间 ）\* CPU 数目

这个公式的线程 cpu 时间是预估的程序单个线程在 cpu 上运行的时间（通常使用 loadrunner测试大量运行次数求出平均值）

**拒绝策略：**

1. AbortPolicy：直接抛出异常，默认策略；  
   2、CallerRunsPolicy：用调用者所在的线程来执行任务；  
   3、DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；  
   4、DiscardPolicy：直接丢弃任务；  
   当然也可以根据应用场景实现 RejectedExecutionHandler 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务

## 7、常见线程安全的并发容器有哪些？

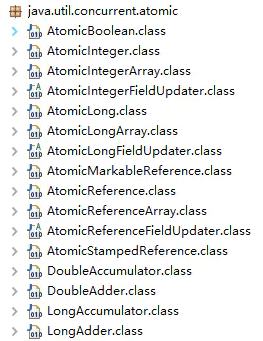
CopyOnWriteArrayList、CopyOnWriteArraySet、ConcurrentHashMap

CopyOnWriteArrayList、CopyOnWriteArraySet采用写时复制实现线程安全

ConcurrentHashMap采用分段锁的方式实现线程安全

## 8、atomic原子类了解多少？原理是什么？

Java 的原子类都存放在并发包 java.util.concurrent.atomic 下，如下图：



**基本类型**

* 使用原子的方式更新基本类型
* AtomicInteger：整形原子类
* AtomicLong：长整型原子类
* AtomicBoolean：布尔型原子类

**数组类型**

* 使用原子的方式更新数组里的某个元素
* AtomicIntegerArray：整形数组原子类
* AtomicLongArray：长整形数组原子类
* AtomicReferenceArray：引用类型数组原子类

**引用类型**

* AtomicReference：引用类型原子类
* AtomicStampedReference：原子更新引用类型里的字段原子类
* AtomicMarkableReference ：原子更新带有标记位的引用类型
* 对象的属性修改类型
* AtomicIntegerFieldUpdater：原子更新整形字段的更新器
* AtomicLongFieldUpdater：原子更新长整形字段的更新器
* AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，以及解决使用 CAS 进行原子更新时可能出现的 ABA 问题

AtomicInteger 类利用 CAS (Compare and Swap) + volatile + native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS 的原理，是拿期望值和原本的值作比较，如果相同，则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是个本地方法，这个方法是用来拿“原值”的内存地址，返回值是 valueOffset；另外，value 是一个 volatile 变量，因此 JVM 总是可以保证任意时刻的任何线程总能拿到该变量的最新值。

## 9、synchronized底层实现是什么？lock底层是什么？有什么区别？

**Synchronized原理：**

方法级的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有monitor（虚拟机规范中用的是管程一词）， 然后再执行方法，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor。

代码块的同步是利用monitorenter和monitorexit这两个字节码指令。它们分别位于同步代码块的开始和结束位置。当jvm执行到monitorenter指令时，当前线程试图获取monitor对象的所有权，如果未加锁或者已经被当前线程所持有，就把锁的计数器+1；当执行monitorexit指令时，锁计数器-1；当锁计数器为0时，该锁就被释放了。如果获取monitor对象失败，该线程则会进入阻塞状态，直到其他线程释放锁。

**Lock原理：**

 Lock的存储结构：一个int类型状态值（用于锁的状态变更），一个双向链表（用于存储等待中的线程）

 Lock获取锁的过程：本质上是通过CAS来获取状态值修改，如果当场没获取到，会将该线程放在线程等待链表中。

 Lock释放锁的过程：修改状态值，调整等待链表。

 Lock大量使用CAS+自旋。因此根据CAS特性，lock建议使用在低锁冲突的情况下。

**Lock与synchronized的区别：**

1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的

2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的     条件下提供一种退出的机制。

3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以    及设置等待时限等方式退出条件队列。

4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

## 10、java类加载过程?

Java类加载需要经历一下几个过程：

**（1）加载**

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。

**（2）验证**

验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。
5. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

**（3）解析**

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

**（4）初始化**

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

## 11、什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

（1）启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。

（2）扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

（3）系统类加载器（system class loader）也叫应用类加载器：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

（4）用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

## 12、简述java内存分配与回收策略以及Minor GC和Major GC（full GC）

**内存分配：**

**（1）栈区**：栈分为java虚拟机栈和本地方法栈

**（2）堆区**：堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。

**（3）方法区**：被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）

**（4）程序计数器**：当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。线程私有的。

**回收策略以及Minor GC和Major GC：**

（1）对象优先在堆的Eden区分配。

（2）大对象直接进入老年代。

（3）长期存活的对象将直接进入老年代。

当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor GC通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生GC的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

# 第五章 MySql数据库篇

## 1、关系数据库中连接池的机制是什么？

前提：为数据库连接建立一个缓冲池。

（1）从连接池获取或创建可用连接

（2）使用完毕之后，把连接返回给连接池

（3）在系统关闭前，断开所有连接并释放连接占用的系统资源

（4）能够处理无效连接，限制连接池中的连接总数不低于或者不超过某个限定值。

其中有几个概念需要大家理解：

最小连接数是连接池一直保持的数据连接。如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费掉。

最大连接数是连接池能申请的最大连接数。如果数据连接请求超过此数，后面的数据连接请求将被加入到等待队列中，这会影响之后的数据库操作。

如果最小连接数与最大连接数相差太大，那么，最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

上面的解释，可以这样理解：数据库池连接数量一直保持一个不少于最小连接数的数量，当数量不够时，数据库会创建一些连接，直到一个最大连接数，之后连接数据库就会等待。

## 2、SQL 的select 语句完整的执行顺序

SQL Select 语句完整的执行顺序：

（1）from 子句组装来自不同数据源的数据；

（2）where 子句基于指定的条件对记录行进行筛选；

（3）group by 子句将数据划分为多个分组；

（4）使用聚集函数进行计算；

（5）使用 having 子句筛选分组；

（6）计算所有的表达式；

（7）select 的字段；

（8）使用order by 对结果集进行排序。

## 3、MySQL的事务

**事务的基本要素（ACID）：**

* 原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位
* 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏 。比如A向B转账，不可能A扣了钱，B却没收到。
* 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
* 持久性（Durability）：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**事务的并发问题：**

* 脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据
* 不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致
* 幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

**小结：**

不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

**MySQL事务隔离级别：**

事务隔离级别 脏读 不可重复读 幻读

读未提交（read-uncommitted） 是 是 是

不可重复读（read-committed） 否 是 是

可重复读（repeatable-read） 否 否 是

串行化（serializable） 否 否 否

## 4、简述在MySQL数据库中MyISAM和InnoDB的区别

**InnoDB存储引擎**

主要面向OLTP(Online Transaction Processing，在线事务处理)方面的应用

**特点：**

行锁设计、支持外键；

**MyISAM存储引擎**

主要面向OLAP(Online Analytical Processing,在线分析处理)方面的应用。

**特点：**

不支持事务，支持表所和全文索引。操作速度快。

## 5、悲观锁和乐观锁的怎么实现？

**悲观锁：**select...for update是MySQL提供的实现悲观锁的方式。

例如：select price from item where id=100 for update

此时在items表中，id为100的那条数据就被我们锁定了，其它的要执行select price from items where id=100 for update的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。MySQL有个问题是select...for update语句执行中所有扫描过的行都会被锁上，因此在MySQL中用悲观锁务必须确定走了索引，而不是全表扫描，否则将会将整个数据表锁住。

**乐观锁：**乐观锁相对悲观锁而言，它认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回错误信息，让用户决定如何去做。

利用数据版本号（version）机制是乐观锁最常用的一种实现方式。一般通过为数据库表增加一个数字类型的 “version” 字段，当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值+1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据，返回更新失败。

举例：

//1: 查询出商品信息

select (quantity,version) from items where id=100;

//2: 根据商品信息生成订单

insert into orders(id,item\_id) values(null,100);

//3: 修改商品的库存

update items set quantity=quantity-1,version=version+1 where id=100 and version=#{version};

## 6、你们公司有哪些数据库设计规范

**（一）基础规范**

1、表存储引擎必须使用InnoD，表字符集默认使用utf8，必要时候使用utf8mb4

解读：

（1）通用，无乱码风险，汉字3字节，英文1字节

（2）utf8mb4是utf8的超集，有存储4字节例如表情符号时，使用它

2、禁止使用存储过程，视图，触发器，Event

解读：

（1）对数据库性能影响较大，互联网业务，能让站点层和服务层干的事情，不要交到数据库层

（2）调试，排错，迁移都比较困难，扩展性较差

3、禁止在数据库中存储大文件，例如照片，可以将大文件存储在对象存储系统，数据库中存储路径

4、禁止在线上环境做数据库压力测试

5、测试，开发，线上数据库环境必须隔离

**（二）命名规范**

1、库名，表名，列名必须用小写，采用下划线分隔

解读：abc，Abc，ABC都是给自己埋坑

2、库名，表名，列名必须见名知义，长度不要超过32字符

解读：tmp，wushan谁知道这些库是干嘛的

3、库备份必须以bak为前缀，以日期为后缀

4、从库必须以-s为后缀

5、备库必须以-ss为后缀

**（三）表设计规范**

1、单实例表个数必须控制在2000个以内

2、单表分表个数必须控制在1024个以内

3、表必须有主键，推荐使用UNSIGNED整数为主键

潜在坑：删除无主键的表，如果是row模式的主从架构，从库会挂住

4、禁止使用外键，如果要保证完整性，应由应用程式实现

解读：外键使得表之间相互耦合，影响update/delete等SQL性能，有可能造成死锁，高并发情况下容易成为数据库瓶颈

5、建议将大字段，访问频度低的字段拆分到单独的表中存储，分离冷热数据

**（四）列设计规范**

1、根据业务区分使用tinyint/int/bigint，分别会占用1/4/8字节

2、根据业务区分使用char/varchar

解读：

（1）字段长度固定，或者长度近似的业务场景，适合使用char，能够减少碎片，查询性能高

（2）字段长度相差较大，或者更新较少的业务场景，适合使用varchar，能够减少空间

3、根据业务区分使用datetime/timestamp

解读：前者占用5个字节，后者占用4个字节，存储年使用YEAR，存储日期使用DATE，存储时间使用datetime

4、必须把字段定义为NOT NULL并设默认值

解读：

（1）NULL的列使用索引，索引统计，值都更加复杂，MySQL更难优化

（2）NULL需要更多的存储空间

（3）NULL只能采用IS NULL或者IS NOT NULL，而在=/!=/in/not in时有大坑

5、使用INT UNSIGNED存储IPv4，不要用char(15)

6、使用varchar(20)存储手机号，不要使用整数

解读：

（1）牵扯到国家代号，可能出现+/-/()等字符，例如+86

（2）手机号不会用来做数学运算

（3）varchar可以模糊查询，例如like ‘138%’

7、使用TINYINT来代替ENUM

解读：ENUM增加新值要进行DDL操作

**（五）索引规范**

1、唯一索引使用uniq\_[字段名]来命名

2、非唯一索引使用idx\_[字段名]来命名

3、单张表索引数量建议控制在5个以内

解读：

（1）互联网高并发业务，太多索引会影响写性能

（2）生成执行计划时，如果索引太多，会降低性能，并可能导致MySQL选择不到最优索引

（3）异常复杂的查询需求，可以选择ES等更为适合的方式存储

4、组合索引字段数不建议超过5个

解读：如果5个字段还不能极大缩小row范围，八成是设计有问题

5、不建议在频繁更新的字段上建立索引

6、非必要不要进行JOIN查询，如果要进行JOIN查询，被JOIN的字段必须类型相同，并建立索引

解读：踩过因为JOIN字段类型不一致，而导致全表扫描的坑么？

7、理解组合索引最左前缀原则，避免重复建设索引，如果建立了(a,b,c)，相当于建立了(a), (a,b), (a,b,c)

**（六）SQL规范**

1、禁止使用select \*，只获取必要字段

解读：

（1）select \*会增加cpu/io/内存/带宽的消耗

（2）指定字段能有效利用索引覆盖

（3）指定字段查询，在表结构变更时，能保证对应用程序无影响

2、insert必须指定字段，禁止使用insert into T values()

解读：指定字段插入，在表结构变更时，能保证对应用程序无影响

3、隐式类型转换会使索引失效，导致全表扫描

4、禁止在where条件列使用函数或者表达式

解读：导致不能命中索引，全表扫描

5、禁止负向查询以及%开头的模糊查询

解读：导致不能命中索引，全表扫描

6、禁止大表JOIN和子查询

7、同一个字段上的OR必须改写问IN，IN的值必须少于50个

8、应用程序必须捕获SQL异常

解读：方便定位线上问题

说明：本规范适用于并发量大，数据量大的典型互联网业务，可直接参考。

## 7、索引的底层实现是什么？什么情况下会索引失效？

InnoDB 存储引擎是用 B+Tree 实现其索引结构

**失效条件：**

1.如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)

　　要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引

2.对于多列索引，不是使用的第一部分，则不会使用索引

3.like查询以%开头

4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引

6. 组合索引要遵循 最左匹配原则

## 8、MySQL性能优化

（1）尽量选择较小的列

（2）将where中用的比较频繁的字段建立索引

（3）select子句中避免使用‘\*’

（4）避免在索引列上使用计算、not in 和<>等操作

（5）当只需要一行数据的时候使用limit 1

（6）保证单表数据不超过200W，适时分割表。针对查询较慢的语句，可以使用explain 来分析该语句具体的执行情况。

（7）避免改变索引列的类型。

（8）选择最有效的表名顺序，from字句中写在最后的表是基础表，将被最先处理，在from子句中包含多个表的情况下，你必须选择记录条数最少的表作为基础表。

（9）避免在索引列上面进行计算。

（10）尽量缩小子查询的结果

## 9、SQL 语句优化案例

**例1：where 子句中可以对字段进行 null 值判断吗？**

可以，比如 select id from t where num is null 这样的 sql 也是可以的。但是最好不要给数据库留NULL，尽可能的使用 NOT NULL 填充数据库。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段，null 不占用空间。可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：select id from t where num= 0。

**例2：如何优化?下面的语句？**

select \* from admin left join log on admin.admin\_id = log.admin\_id where log.admin\_id>10

优化为：select \* from (select \* from admin where admin\_id>10) T1 lef join log on T1.admin\_id = log.admin\_id。

使用 JOIN 时候，应该用小的结果驱动大的结果（left join 左边表结果尽量小如果有条件应该放到左边先处理， right join 同理反向），同时尽量把牵涉到多表联合的查询拆分多个 query（多个连表查询效率低，容易到之后锁表和阻塞）。

**例3：limit 的基数比较大时使用 between**

例如：select \* from admin order by admin\_id limit 100000,10

优化为：select \* from admin where admin\_id between 100000 and 100010 order by admin\_id。

**例4：尽量避免在列上做运算，这样导致索引失效**

例如：select \* from admin where year(admin\_time)>2014

优化为： select \* from admin where admin\_time> '2014-01-01′

## 10、常见面试sql

**例1：**

用一条SQL语句查询出每门课都大于80分的学生姓名

name kecheng fenshu  
张三 语文 81  
张三 数学     75  
李四 语文     76  
李四 数学    90  
王五 语文     81  
王五 数学     100  
王五 英语     90  
**答1：**

**select** **distinct** **name** **from** **table** **where** **name** not in (**select** **distinct** **name** **from** **table** **where** fenshu<=80)    
**答2：**

**select** **name** **from** **table** **group** **by** **name** **having** **min**(fenshu)>80

**例2：**

学生表 如下:  
自动编号 学号 姓名 课程编号 课程名称 分数  
1      2005001  张三  0001    数学   69  
2      2005002  李四  0001    数学   89  
3      2005001  张三   0001    数学   69  
删除除了自动编号不同，其他都相同的学生冗余信息  
**答：**

**delete** tablename **where** 自动编号 not in(**select** **min**(自动编号) **from** tablename **group** **by**学号, 姓名, 课程编号, 课程名称, 分数)

**例3：**

一个叫team的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球队，现在四个球队进行比赛，用一条sql语句显示所有可能的比赛组合.

**答：**

1. **select** a.**name**, b.**name**
2. **from** team a, team b
3. **where** a.**name** < b.**name**

**例4：**

怎么把这样一个表  
year month amount  
1991   1     1.1  
1991   2     1.2  
1991   3     1.3  
1991   4     1.4  
1992   1     2.1  
1992   2     2.2  
1992   3     2.3  
1992   4     2.4  
查成这样一个结果  
year m1 m2 m3 m4  
1991 1.1 1.2 1.3 1.4  
1992 2.1 2.2 2.3 2.4   
**答：**

1. **select** year,
2. (**select** amount **from** aaa m **where** month=1 and m.year=aaa.year) **as** m1,
3. (**select** amount **from** aaa m **where** month=2 and m.year=aaa.year) **as** m2,
4. (**select** amount **from** aaa m **where** month=3 and m.year=aaa.year) **as** m3,
5. (**select** amount **from**  aaa m **where** month=4 and m.year=aaa.year) **as** m4
6. **from** aaa **group** **by** year

**例5：**

说明：复制表(只复制结构,源表名：a新表名：b)

**答：**  
SQL:

**select** \* **into** b **from** a **where** 1<>1 (where1=1，拷贝表结构和数据内容)

ORACLE:

1. **create** **table** b
2. **As**
3. **Select** \* **from** a **where** 1=2

[<>（不等于）(SQL Server Compact)

比较两个表达式。 当使用此运算符比较非空表达式时，如果左操作数不等于右操作数，则结果为 TRUE。 否则，结果为 FALSE。]

**例6：**

原表：  
courseid coursename score  
1 java 70  
2 oracle 90  
3 xml 40  
4 jsp 30  
5 servlet 80

为了便于阅读,查询此表后的结果显式如下(及格分数为60):  
courseid coursename score mark  
1 java 70 pass  
2 oracle 90 pass  
3 xml 40 fail  
4 jsp 30 fail  
5 servlet 80 pass  
写出此查询语句

**答：**

**select** courseid, coursename ,score ,if(score>=60, "pass","fail")  **as** mark **from** course

**例7：**

表名：购物信息

购物人 商品名称 数量

A 甲 2

B 乙 4

C 丙 1

A 丁 2

B 丙 5

给出所有购入商品为两种或两种以上的购物人记录

**答：**

**select** \* **from** 购物信息 **where** 购物人 in (**select** 购物人 **from** 购物信息 **group** **by** 购物人 **having** count(\*) >= 2);

**例8：**

info 表

date result

2005-05-09 win

2005-05-09 lose

2005-05-09 lose

2005-05-09 lose

2005-05-10 win

2005-05-10 lose

2005-05-10 lose

如果要生成下列结果, 该如何写sql语句?

date win lose

2005-05-09 2 2

2005-05-10 1 2

**答1：**

**select** **date**, sum(case **when** result = "win" **then** 1 **else** 0 **end**) **as** "win", sum(case **when** result = "lose" **then** 1 **else** 0 **end**) **as** "lose" **from** info **group** **by** **date**;

**答2：**

1. **select** a.**date**, a.result **as** win, b.result **as** lose
2. **from**
3. (**select** **date**, count(result) **as** result **from** info **where** result = "win" **group** **by** **date**) **as** a
4. join
5. (**select** **date**, count(result) **as** result **from** info **where** result = "lose" **group** **by** **date**) **as** b
6. **on** a.**date** = b.**date**;

# 第六章 Java Web篇

## 1、http 的长连接和短连接

HTTP 协议有 HTTP/1.0 版本和 HTTP/1.1 版本。HTTP1.1 默认保持长连接（HTTP persistent connection，也翻译为持久连接），数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

在 HTTP/1.0 中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。从 HTTP/1.1 起，默认使用的是长连接，用以保持连接特性。

## 2、http 常见的状态码有哪些？

200 OK //客户端请求成功

301 Moved Permanently（永久移除)，请求的 URL 已移走。Response 中应该包含一个 Location URL, 说明资源现在所处的位置

302 found 重定向

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

## 3、GET 和POST 的区别？

（1）GET 请求的数据会附在URL 之后（就是把数据放置在 HTTP 协议头中），以?分割URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置在是 HTTP 包的包体中。

（2）GET 方式提交的数据最多只能是 1024 字节，理论上POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：“GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对URL 长度的限制是2083 字节(2K+35)。对于其他浏览器，如Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持。

（3）POST 的安全性要比GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。

Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method

默认为"GET"，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！

## 4、Cookie 和Session 的区别

Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和session 的不同点：

（1）无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

（2）在存储的数据量方面：session 能够存储任意的java 对象，cookie 只能存储 String 类型的对象。

## 5、什么是jsp，什么是Servlet？jsp 和Servlet 有什么区别？

jsp 本质上就是一个Servlet，它是 Servlet 的一种特殊形式（由 SUN 公司推出），每个 jsp 页面都是一个servlet实例。

Servlet 是由 Java 提供用于开发 web 服务器应用程序的一个组件，运行在服务端，由 servlet 容器管理，用来生成动态内容。一个 servlet 实例是实现了特殊接口 Servlet 的 Java 类，所有自定义的 servlet 均必须实现 Servlet 接口。

**区别：**

jsp 是 html 页面中内嵌的Java 代码，侧重页面显示；

Servlet 是 html 代码和 Java 代码分离，侧重逻辑控制，mvc 设计思想中jsp 位于视图层，servlet 位于控制层

Jsp 运行机制：如下图



JVM 只能识别 Java 类，并不能识别 jsp 代码！web 容器收到以.jsp 为扩展名的 url 请求时，会将访问请求交给tomcat 中 jsp 引擎处理，每个 jsp 页面第一次被访问时，jsp 引擎将 jsp 代码解释为一个 servlet 源程序，接着编译servlet 源程序生成.class 文件，再有 web 容器 servlet 引擎去装载执行servlet 程序，实现页面交互。

## 6、servlet生命周期

Servlet 加载—>实例化—>服务—>销毁。

**生命周期详解：**

**init（）：**

在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。

**service（）：**

它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。

**destroy（）：**

仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

**如何与Tomcat 结合工作步骤：**

（1）Web Client 向Servlet容器（Tomcat）发出Http请求

（2）Servlet容器接收Web Client的请求

（3）Servlet容器创建一个HttpRequest对象，将Web Client请求的信息封装到这个对象中。

（4）Servlet容器创建一个HttpResponse对象

（5）Servlet容器调用HttpServlet对象的service方法，把HttpRequest对象与HttpResponse对象作为参数传给HttpServlet 对象。

（6）HttpServlet调用HttpRequest对象的有关方法，获取Http请求信息。

（7）HttpServlet调用HttpResponse对象的有关方法，生成响应数据。

## 7、servlet特性

单例多线程

# 第七章 Java框架篇

## 1、简单的谈一下SpringMVC的工作流程？

* 用户发送请求至前端控制器DispatcherServlet
* DispatcherServlet收到请求调用HandlerMapping处理器映射器。
* 处理器映射器找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
* DispatcherServlet调用HandlerAdapter处理器适配器
* HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
* Controller执行完成返回ModelAndView
* HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
* DispatcherServlet将ModelAndView传给ViewReslover视图解析器
* ViewReslover解析后返回具体View
* DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
* DispatcherServlet响应用户

## 2、说出Spring 或者 Springmvc中常用的5个注解，并解释含义

@Component 基本注解，标识一个受Spring管理的组件

@Controller 标识为一个表示层的组件

@Service 标识为一个业务层的组件

@Repository 标识为一个持久层的组件

@Autowired 自动装配

@Qualifier(“”) 具体指定要装配的组件的id值

@RequestMapping() 完成请求映射

@PathVariable 映射请求URL中占位符到请求处理方法的形参

只要说出机几个注解并解释含义即可，如上答案只做参考

## 3、简述Springmvc中如何返回JSON数据

Step1：在项目中加入json转换的依赖，例如jackson，fastjson，gson等

Step2：在请求处理方法中将返回值改为具体返回的数据的类型， 例如数据的集合类List<Employee>等

Step3：在请求处理方法上使用@ResponseBody注解

## 4、谈谈你对Spring 的理解

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的JavaBean 实现以前只有EJB 才能实现的功能。Spring 是一个 IOC 和 AOP 容器框架。

Spring 容器的主要核心是：

控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用spring 提供的对象就可以了，这是控制反转的思想。

依赖注入（DI），spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。

面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用CGLIB 方式实现动态代理。

## 5、Spring中常用的设计模式

（1）代理模式——spring 中两种代理方式，若目标对象实现了若干接口，spring 使用jdk 的java.lang.reflect.Proxy类代理。若目标兑现没有实现任何接口，spring 使用 CGLIB 库生成目标类的子类。

（2）单例模式——在 spring 的配置文件中设置 bean 默认为单例模式。

（3）模板方式模式——用来解决代码重复的问题。

比如：RestTemplate、JmsTemplate、JpaTemplate

（4）工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。Spring 中使用 beanFactory 来创建对象的实例。

## 6、介绍一下Spring bean 的生命周期、注入方式和作用域

**Bean的生命周期**

（1）默认情况下，IOC容器中bean的生命周期分为五个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.
* 使用
* IOC容器关闭时， 销毁Bean对象.

（2）当加入了Bean的后置处理器后，IOC容器中bean的生命周期分为七个阶段:

* 调用构造器 或者是通过工厂的方式创建Bean对象
* 给bean对象的属性注入值
* 执行Bean后置处理器中的 postProcessBeforeInitialization
* 调用初始化方法，进行初始化， 初始化方法是通过init-method来指定的.
* 执行Bean的后置处理器中 postProcessAfterInitialization
* 使用
* IOC容器关闭时， 销毁Bean对象

只需要回答出第一点即可，第二点也回答可适当 加分。

**注入方式：**

通过 setter 方法注入

通过构造方法注入

**Bean的作用域**

总共有四种作用域:

* Singleton 单例的
* Prototype 原型的
* Request
* Session

## 7、请描述一下Spring 的事务管理

（1）声明式事务管理的定义：用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。这样的好处是，事务管理不侵入开发的组件，具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可，这样维护起来极其方便。

基于 TransactionInterceptor 的声明式事务管理：两个次要的属性： transactionManager，用来指定一个事务治理器， 并将具体事务相关的操作请托给它； 其他一个是 Properties 类型的transactionAttributes 属性，该属性的每一个键值对中，键指定的是方法名，方法名可以行使通配符， 而值就是表现呼应方法的所运用的事务属性。

（2）基于 @Transactional 的声明式事务管理：Spring 2.x 还引入了基于 Annotation 的体式格式，具体次要触及@Transactional 标注。@Transactional 可以浸染于接口、接口方法、类和类方法上。算作用于类上时，该类的一切public 方法将都具有该类型的事务属性。

（3）编程式事物管理的定义：在代码中显式挪用 beginTransaction()、commit()、rollback()等事务治理相关的方法， 这就是编程式事务管理。Spring 对事物的编程式管理有基于底层 API 的编程式管理和基于 TransactionTemplate 的编程式事务管理两种方式。

## 8、MyBatis中 #{}和${}的区别是什么？

#{}是预编译处理，${}是字符串替换；

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理${}时，就是把${}替换成变量的值；

使用#{}可以有效的防止SQL注入，提高系统安全性。

## 9、Mybatis 中一级缓存与二级缓存？

（1）MyBatis的缓存分为一级缓存和 二级缓存。

一级缓存是SqlSession级别的缓存，默认开启。

二级缓存是NameSpace级别(Mapper)的缓存，多个SqlSession可以共享，使用时需要进行配置开启。

（2）缓存的查找顺序：二级缓存 => 一级缓存 => 数据库

## 10、MyBatis如何获取自动生成的(主)键值?

在<insert>标签中使用 useGeneratedKeys 和 keyProperty 两个属性来获取自动生成的主键值。

示例:

1. **<insert** id=”insertname” usegeneratedkeys=”true” keyproperty=”id”**>**
2. insert into names (name) values (#{name})
3. **</insert>**

## 11、简述Mybatis的动态SQL，列出常用的6个标签及作用

动态SQL是MyBatis的强大特性之一 基于功能强大的OGNL表达式。

动态SQL主要是来解决查询条件不确定的情况，在程序运行期间，根据提交的条件动态的完成查询

常用的标签:

<if> : 进行条件的判断

<where>：在<if>判断后的SQL语句前面添加WHERE关键字，并处理SQL语句开始位置的AND 或者OR的问题

<trim>：可以在SQL语句前后进行添加指定字符 或者去掉指定字符.

<set>: 主要用于修改操作时出现的逗号问题

<choose> <when> <otherwise>：类似于java中的switch语句.在所有的条件中选择其一

<foreach>：迭代操作

## 12、Mybatis 如何完成MySQL的批量操作,举例说明

MyBatis完成MySQL的批量操作主要是通过<foreache>标签来拼装相应的SQL语句.

例如:

1. **<insert** id="insertBatch" **>**
2. insert into tbl\_employee(last\_name,email,gender,d\_id) values
3. **<foreach** collection="emps" item="curr\_emp" separator=","**>**
4. (#{curr\_emp.lastName},#{curr\_emp.email},#{curr\_emp.gender},#{curr\_emp.dept.id})
5. **</foreach>**
6. **</insert>**

## 13、谈谈怎么理解SpringBoot框架？

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。



**Spring Boot的优点**

* 独立运行

Spring Boot而且内嵌了各种servlet容器，Tomcat、Jetty等，现在不再需要打成war包部署到容器中，Spring Boot只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

* 简化配置

spring-boot-starter-web启动器自动依赖其他组件，简少了maven的配置。除此之外，还提供了各种启动器，开发者能快速上手。

* 自动配置

Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

* 无代码生成和XML配置

Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是Spring4.x的核心功能之一。

* 应用监控

Spring Boot提供一系列端点可以监控服务及应用，做健康检测。

**Spring Boot缺点：**

Spring Boot虽然上手很容易，但如果你不了解其核心技术及流程，所以一旦遇到问题就很棘手，而且现在的解决方案也不是很多，需要一个完善的过程。

## 14、Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

* @SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。
* @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，
  + 如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
* @ComponentScan：Spring组件扫描。

## 15、Spring Boot 自动配置原理是什么？

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

@EnableAutoConfiguration是实现自动配置的注解

@Configuration表示这是一个配置文件

具体参考文档：

<https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484365&idx=1&sn=a4ab1d977d6b03bf122b4d596d7ee1ab&scene=21>

## 16、SpringBoot配置文件有哪些？怎么实现多环境配置？

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

**bootstrap配置文件的特性：**

* boostrap 由父 ApplicationContext 加载，比 applicaton 优先加载
* boostrap 里面的属性不能被覆盖

**bootstrap 配置文件有以下几个应用场景：**

* 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
* 一些固定的不能被覆盖的属性；
* 一些加密/解密的场景；

提供多套配置文件，如：

applcation.properties  
application-dev.properties  
application-test.properties  
application-prod.properties

运行时指定具体的配置文件，具体请看这篇文章《[Spring Boot Profile 不同环境配置](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247484369&idx=1&sn=1155fccb4fef1db88cb76fd17b1756d7&scene=21#wechat_redirect)》。

## 17、Spring Boot和Spring Cloud是什么关系

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架； Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现，必须基于Spring Boot开发。

可以单独使用Spring Boot开发项目，但是Spring Cloud离不开 Spring Boot。

## 18、SpringCloud都用过哪些组件？介绍一下作用

**经常用的组件：**

服务发现——Netflix Eureka实现服务治理（服务注册与发现）

远程调用——Feign 实现服务间的远程调用

客服端负载均衡——Netflix Ribbon主要提供客户侧的软件负载均衡算法。

断路器——Netflix Hystrix断路器，保护系统，控制故障范围。

服务网关——Gateway网关，路由，负载均衡等多种作用

分布式配置——Spring Cloud Config配置管理

## 19、Eureka和zookeeper的区别

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

**Zookeeper保证CP**

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

**Eureka保证AP**

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

**总结**

Eureka作为单纯的服务注册中心来说要比zookeeper更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过Eureka目前1.X版本的实现是基于servlet的java web应用，它的极限性能肯定会受到影响。期待正在开发之中的2.X版本能够从servlet中独立出来成为单独可部署执行的服务。

# 第八章 Redis数据库篇

## 1、介绍下Redis？redis有哪些数据类型？

**Redis全称（Remote Dictionary Server）**本质上是一个Key-Value类型的内存数据库，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。 Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。 Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

**常用数据类型如下：**

|  |  |
| --- | --- |
| string | 字符串（一个字符串类型最大存储容量为512M） |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset(sorted set） | 带分数的set |

## 2、为什么Redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。 如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

## 3、Redis提供了哪几种持久化方式？

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

你也可以同时开启两种持久化方式，在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

**（1）RDB持久化：**

每隔一段时间，将内存中的数据集写到磁盘

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

**保存策略：**

save 9 00 1 900 秒内如果至少有 1 个 key 的值变化，则保存

save 300 10 300 秒内如果至少有 10 个 key 的值变化，则保存

save 60 1 0000 60 秒内如果至10000 个 key 的值变化，则保存

**（2）AOF 持久化: 以日志形式记录每个更新（(总结、改）操作**

Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。

**保存策略：**

appendfsync always：每次产生一条新的修改数据的命令都执行保存操作；效率低，但是安全！

appendfsync everysec：每秒执行一次保存操作。如果在未保存当前秒内操作时发生了断电，仍然会导致一部分数据丢失（即1秒钟的数据）。

appendfsync no：从不保存，将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

**缺点：**

1 比起RDB占用更多的磁盘空间

2 恢复备份速度要慢

3 每次读写都同步的话，有一定的性能压力

4 存在个别Bug，造成恢复不能

**（3）选择策略：**

可读的日志文本，通过操作AOF

官方推荐：

如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

## 4、redis是单线程的，为什么那么快？

1)完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

2)数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的

3)采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

4)使用多路I/O复用模型，非阻塞IO

5)使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

## 5、缓存穿透怎么解决？

**缓存系统定义：**

按照KEY去查询VALUE，当KEY对应的VALUE一定不存在的时候并对KEY并发请求量很大的时候，就会对后端造成很大的压力。（查询一个必然不存在的数据。比如文章表，查询一个不存在的id，每次都会访问DB，如果有人恶意破坏，很可能直接对DB造成影响。）

由于缓存不命中，每次都要查询持久层。从而失去缓存的意义。

**解决方法：**

（1）缓存层缓 存空值。

缓存太多空值，占用更多空间。（优化：给个空值过 期时间）

存储层更新代码了，缓存层还是空值。（优化：后台设置时主动删除空值，并缓存把值进去）

（2）将数据库中所有的查询条件，放布隆过滤器中。当一个查询请求来临的时候，先经过布隆过滤器进行查，如果请求存在这个条件中，那么继续执行，如果不在，直接丢弃。

**备注 ：**

比如数据库中有10000个条件，那么布隆过滤器的容量size设置的要稍微比10000大一些，比如12000.

对于误判率的设置，根据实际项目，以及硬件设施来具体定。但一定不能设置为0，并且误判率设置的越小，哈希函数跟数组长度都会更多跟更长，那么对硬件，内存中间的要求就会相应 的高

private st atic BloomFilter<Inte ger> bloomFi lt er = BloomFilter.create(Funnels.integerFue l(), s ize, 000 01) ;

有了siz跟误判率，那么布隆过滤器会产相应的哈希函数跟数组。

综上：我们可以利用布隆过滤器，将redis缓存击穿制在一个可容的范围内。

## 6、哨兵模式是什么样的？

如果Master异常，则会进行Master-Slave切换，将其中一Slae作为Master，将之前的Master作为Slave

**下线：**

①主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个redis服务器做出的下线判断。

②客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对Master Server做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，然后开启failover.

**工作原理：**

（1）每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令 ；

（2）如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；

（3）如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态；

（4）当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 ；

（5）在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令

（6）当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次 ；

（7）若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除；

若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除；

## 7、Redis常见性能问题和解决方案？

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

## 8、MySQL里有大量数据，如何保证redis中的数据都是热点数据？

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

**数据淘汰策略：**

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

## 9、Redis集群方案应该怎么做？都有哪些方案？

（1）twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通redis无任何区别，设置好它下属的多个redis实例后，使用时在本需要连接redis的地方改为连接twemproxy，它会以一个代理的身份接收请求并使用一致性hash算法，将请求转接到具体redis，将结果再返回twemproxy。使用方式简便(相对redis只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy自身单端口实例的压力，使用一致性hash后，对redis节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

（2）codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新hash节点。

（3）redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

（4）在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key 进行hash计算，然后去对应的redis实例操作数据。 这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

## 10、说说Redis哈希槽的概念？

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

## 11、Redis有哪些适合的场景？

（1）会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

（2）全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

（4）排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

（5）发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

# 第九章 分布式技术篇

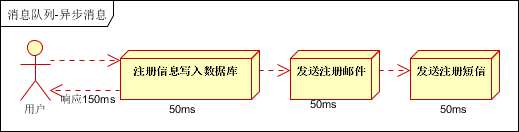
## 1、消息队列的作用与使用场景？

消息队列在实际应用中常用的使用场景。异步处理，应用解耦，流量削锋和消息通讯四个场景

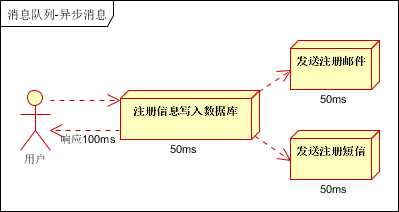
2.1异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式

（1）串行方式：将注册信息写入[数据库](http://lib.csdn.net/base/mysql)成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间

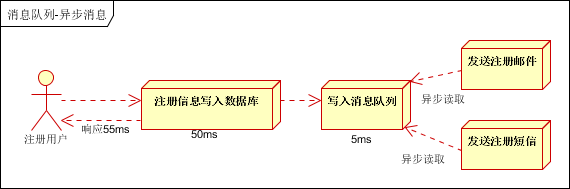


假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次（1000/150）。并行方式处理的请求量是10次（1000/100）

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

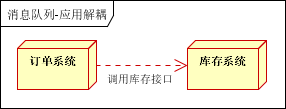
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍

2.2应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图

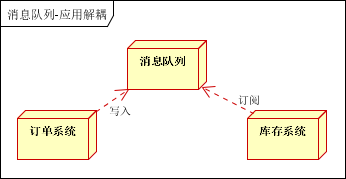


传统模式的缺点：

假如库存系统无法访问，则订单减库存将失败，从而导致订单失败

订单系统与库存系统耦合

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

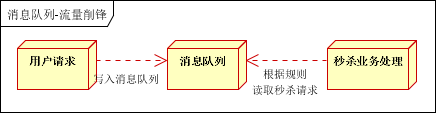
2.3流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

可以控制活动的人数

可以缓解短时间内高流量压垮应用



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面

秒杀业务根据消息队列中的请求信息，再做后续处理

2.4日志处理

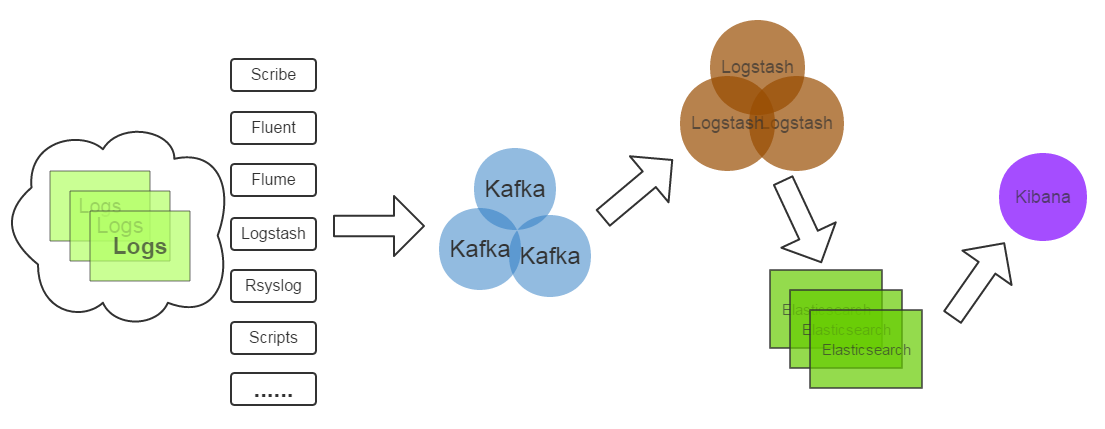
日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

日志采集客户端，负责日志数据采集，定时写受写入Kafka队列

Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

以下是新浪kafka日志处理应用案例：转自（http://cloud.51cto.com/art/201507/484338.htm）



(1)Kafka：接收用户日志的消息队列

(2)Logstash：做日志解析，统一成JSON输出给Elasticsearch

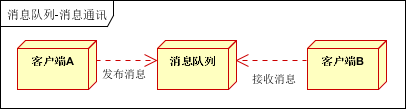
(3)Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能

(4)Kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ELK stack的重要原因

2.5消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等

点对点通讯：



客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

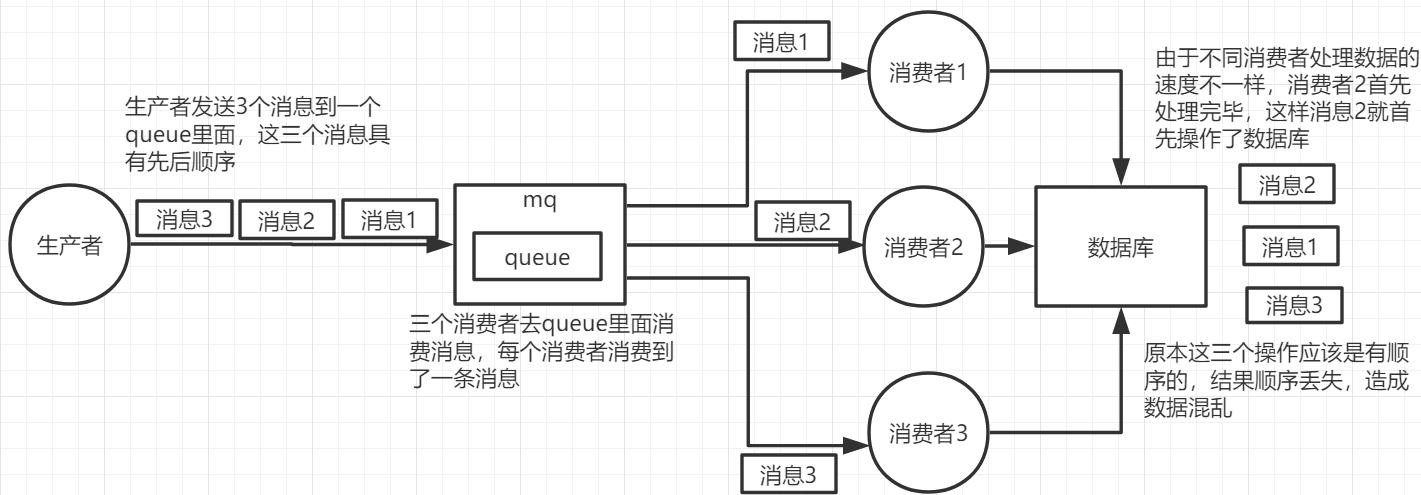
以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

## RabbitMQ如何保证消息的顺序性？

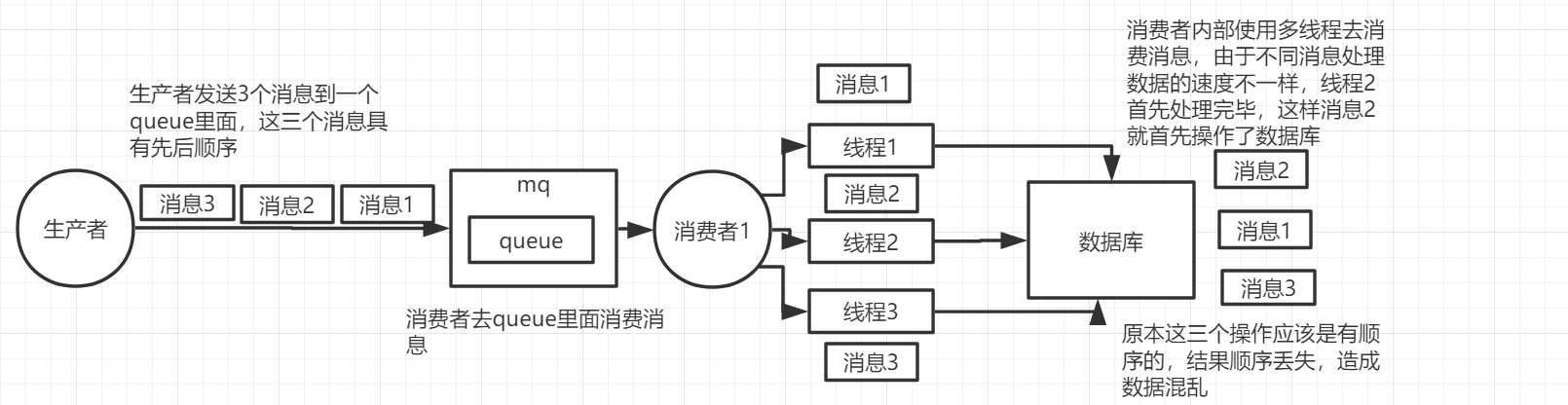
消息队列中的若干消息如果是对同一个数据进行操作，这些操作具有前后的关系，必须要按前后的顺序执行，否则就会造成数据异常。举例：  
比如通过mysql binlog进行两个数据库的数据同步，由于对数据库的数据操作是具有顺序性的，如果操作顺序搞反，就会造成不可估量的错误。比如数据库对一条数据依次进行了 插入->更新->删除操作，这个顺序必须是这样，如果在同步过程中，消息的顺序变成了 删除->插入->更新，那么原本应该被删除的数据，就没有被删除，造成数据的不一致问题。

举例场景：

RabbitMQ：①一个queue，有多个consumer去消费，这样就会造成顺序的错误，consumer从MQ里面读取数据是有序的，但是每个consumer的执行时间是不固定的，无法保证先读到消息的consumer一定先完成操作，这样就会出现消息并没有按照顺序执行，造成数据顺序错误。

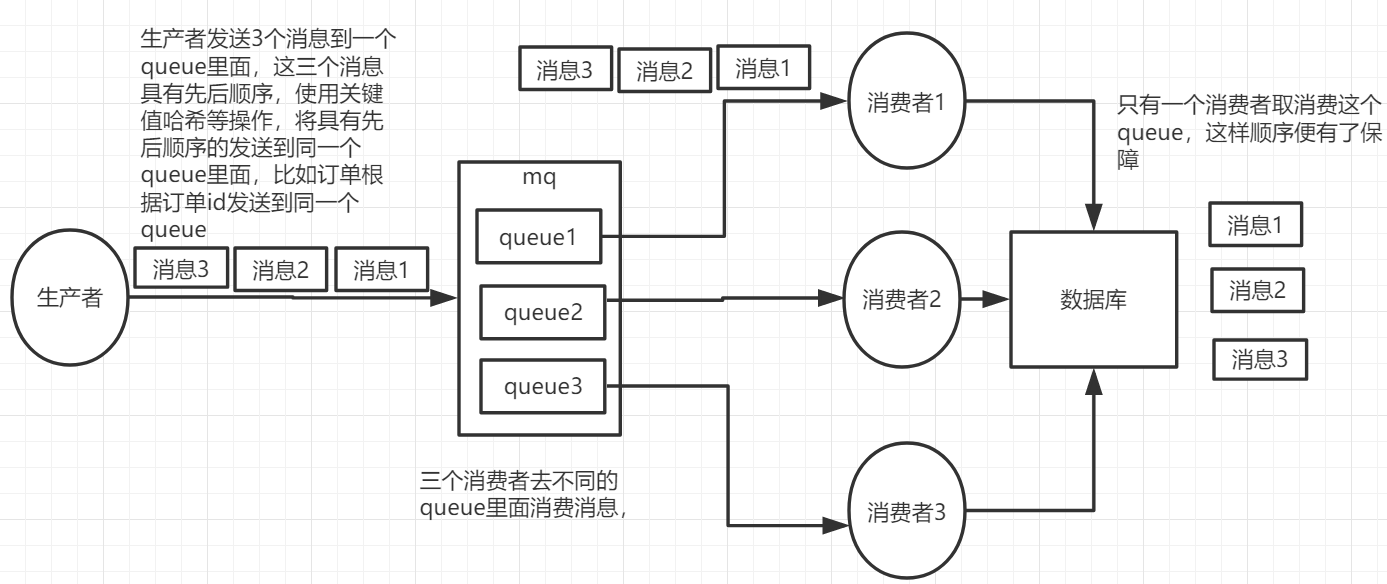


②一个queue对应一个consumer，但是consumer里面进行了多线程消费，这样也会造成消息消费顺序错误。



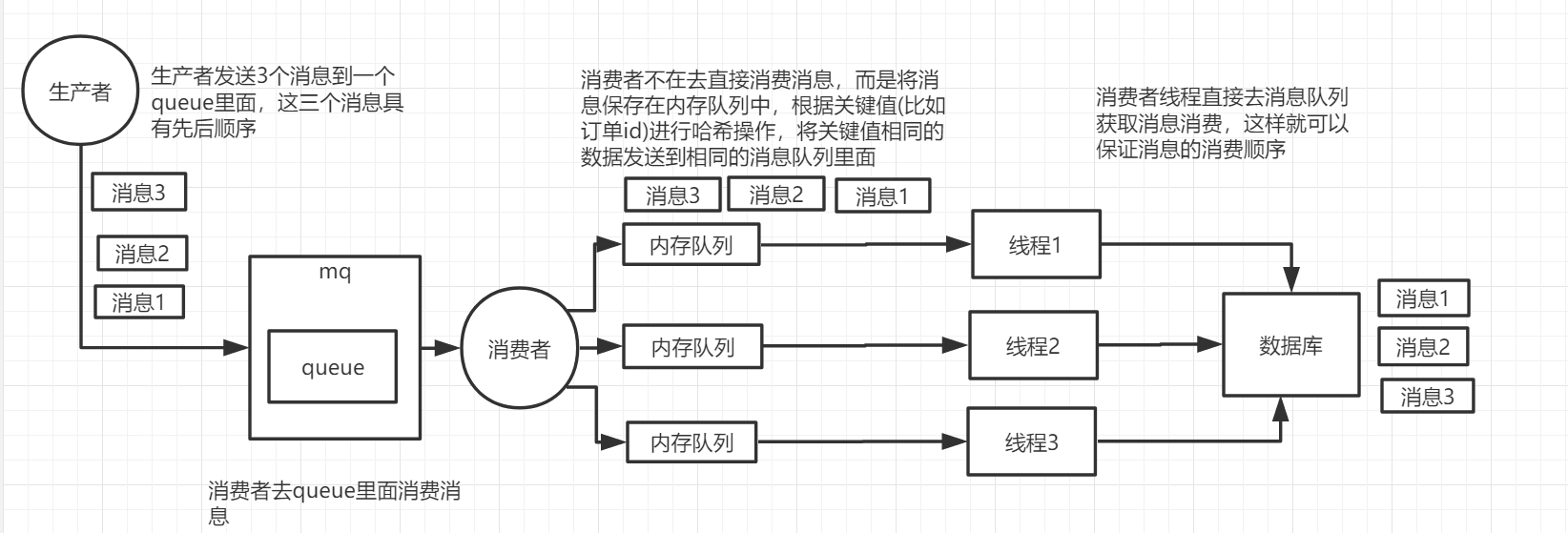
解决方案：

①拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；这样也会造成吞吐量下降，可以在消费者内部采用多线程的方式取消费。



一个queue对应一个consumer

②或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理

一个queue对应一个consumer，采用多线程

## 3、如何使用RabbitMQ解决分布式事务？

在互联网应用中，基本都会有用户注册的功能。在注册的同时，我们会做出如下操作：

收集用户录入信息，保存到数据库向用户的手机或邮箱发送验证码等等…

如果是传统的集中式架构，实现这个功能非常简单：开启一个本地事务，往本地数据库中插入一条用户数据，发送验证码，提交事物。

但是在分布式架构中，用户和发送验证码是两个独立的服务，它们都有各自的数据库，那么就不能通过本地事物保证操作的原子性。这时我们就需要用到 RabbitMQ（消息队列）来为我们实现这个需求。

在用户进行注册操作的时候，我们为该操作创建一条消息，当用户信息保存成功时，把这条消息发送到消息队列。验证码系统会监听消息，一旦接受到消息，就会给该用户发送验证码。

## 4、RabbitMQ如何确保消息的确定性？

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

## 5、如何防止RabbitMQ消息重复消费？

解决方法很简单：增加消息状态表。通俗来说就是一个账本，用来记录消息的处理状态，每次处理消息之前，都去状态表中查询一次，如果已经有相同的消息存在，那么不处理，可以防止重复消费。

## 6、什么是ElasticSearch？ 怎么解决中文分词？

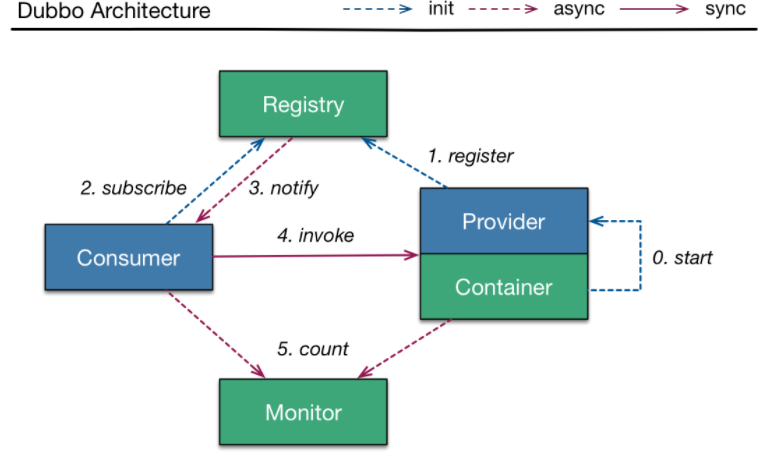
Elasticsearch是一个基于Lucene的搜索引擎。它提供了具有HTTP Web界面和无架构JSON文档的分布式，多租户能力的全文搜索引擎。Elasticsearch是用Java开发的，根据Apache许可条款作为开源发布。

采用IK中文分词器插件进行中文分词处理。

## 7、Elasticsearch中的倒排索引是什么？

倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。

## 8、Dubbo执行流程是什么样的？各个组件作用



它的架构主要有五个角色/核心组件，分为是Container（容器）、Provider（服务的提供方）、Registry（注册中心）、Consumer（服务的消费方）、Monitor（监控中心）。

容器主要负责启动、加载、运行服务提供者；

同时服务提供者在启动时，向注册中心注册自己提供的服务；

消费者向注册中心订阅自己的服务；

注册中心返回服务提供者列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者；

对于服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另外一台调用；

服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心；

**注册中心：**

1. 注册中心只负责地址的注册和查找，相当于我们的目录服务，只有在容器启动时，服务提供者和调用者与注册中心交互，整个过程，注册中心不参与数据传输，不转发请求，压力较小 （“两不一小”）；
2. 注册中心宕机问题：（Dubbo挂了怎么办？）

注册中心会部署集群，

如果集群中的任意一台宕机之后，将自动切换到另外一台，不会影响已运行的提供者和消费者；

如果集群中所有注册中心全部宕机之后，服务提供者和服务消费者仍能通过本地缓存通讯；注册中心仍能通过缓存提供服务列表查询，但不能注册新服务；

**监控中心：**

1. 监控中心负责统计各服务调用次数、调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
2. 监控中心宕机不影响使用，只是丢失部分的采样数据；
3. dubbo-admin可以通过监控中心的可视化界面，进行禁止服务和截止消费者（大量恶意访问的ip）；

## 9、Dubbo 的容错机制有哪些？

（1）Failover Cluster 模 式

失败自动切换，当出现失败，重试其它服务器。(默认) 2）Failfast Cluster

快速失败，只发起一次调用，失败立即报错。 通常用于非幂等性的写操作，比如新增记录。

（2）Failsafe Cluster

失败安全，出现异常时，直接忽略。 通常用于写入审计日志等操作。

（3）Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。 通常用于消息通知操作。

（4）Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks=”2”来设置最大并行数。

（5）Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0 开始支持) 通常用于通知所有提供者更新缓存或日志等本地资源信息。

总结： 在实际应用中查询语句容错策略建议使用默认Failover Cluster ，而增删改建议使用 Failfast Cluster 或者 使用 Failover Cluster（retries=”0”） 策略 防止出现数据 重复添加等等其它问题！建议在设计接口时候把查询接口方法单独做一个接口提供查询。

## 10、使用dubbo 遇到过哪些问题？

（1）增加提供服务版本号和消费服务版本号

这个具体来说不算是一个问题,而是一种问题的解决方案,在我们的实际工作中会面临各种环境资源短缺的问题,也是很实际的问题,刚开始我们还可以提供一个服务进行相关的开发和测试,但是当有多个环境多个版本,多个任务的时候就不满足我们的需求,这时候我们可以通过给提供方增加版本的方式来区分.这样能够剩下很多的物理资源,同时为今后更换接口定义发布在线时，可不停机发布，使用版本号.

引用只会找相应版本的服务,例如：

<dubbo:serviceinterface=“com.xxx.XxxService”ref=“xxxService” version=“1.0” />

<dubbo:referenceid=“xxxService”interface=“com.xxx.XxxService” version=“1.0”/>

（2）dubbo reference 注解问题

@Reference 只能在 springbean 实例对应的当前类中使用，暂时无法在父类使用；如果确实要在父类声明一个引用，可通过配置文件配置 dubbo:reference，然后在需要引用的地方跟引用 springbean 一样就可以了.

（3）出现 RpcException: No provider available for remote service 异常，表示没有可用的服务提供者

* 检查连接的注册中心是否正确
* 到注册中心查看相应的服务提供者是否存在
* 检查服务提供者是否正常运行

（4）服务提供者没挂，但在注册中心里看不到

首先，确认服务提供者是否连接了正确的注册中心，不只是检查配置中的注册中心地址，而且要检查实际的网络连接。

其次，看服务提供者是否非常繁忙，比如压力测试，以至于没有CPU 片段向注册中心发送心跳，这种情况，减小压力，将自动恢复。

## 11、ZooKeeper是什么？

ZooKeeper是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于Zookeeper实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master选举、分布式锁和分布式队列等功能。

Zookeeper保证了如下分布式一致性特性：

* 顺序一致性
* 原子性
* 单一视图
* 可靠性
* 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的zookeeper机器来处理。对于写请求，这些请求会同时发给其他zookeeper机器并且达成一致后，请求才会返回成功。因此，随着zookeeper的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是zookeeper中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个zookeeper最新的zxid。

## 12、Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper允许客户端向服务端的某个Znode注册一个Watcher监听，当服务端的一些指定事件触发了这个Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据Watcher通知状态和事件类型做出业务上的改变。

**工作机制：**

* 客户端注册watcher
* 服务端处理watcher
* 客户端回调watcher

**Watcher特性总结：**

（1）一次性  
无论是服务端还是客户端，一旦一个Watcher被触发，Zookeeper都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

（2）客户端串行执行  
客户端Watcher回调的过程是一个串行同步的过程。

（3）轻量

Watcher通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

客户端向服务端注册Watcher的时候，并不会把客户端真实的Watcher对象实体传递到服务端，仅仅是在客户端请求中使用boolean类型属性进行了标记。

watcher event异步发送watcher的通知事件从server发送到client是异步的，这就存在一个问题，不同的客户端和服务器之间通过socket进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于Zookeeper本身提供了ordering guarantee，即客户端监听事件后，才会感知它所监视znode发生了变化。所以我们使用Zookeeper不能期望能够监控到节点每次的变化。Zookeeper只能保证最终的一致性，而无法保证强一致性。

注册watcher getData、exists、getChildren

触发watcher create、delete、setData

当一个客户端连接到一个新的服务器上时，watch将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到watch的。而当client重新连接时，如果需要的话，所有先前注册过的watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch可能会丢失：对于一个未创建的znode的exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个watch事件可能会被丢失。

## 13、ZooKeeper分布式锁的实现原理

使用zookeeper创建临时序列节点来实现分布式锁，适用于顺序执行的程序，大体思路就是创建临时序列节点，找出最小的序列节点，获取分布式锁，程序执行完成之后此序列节点消失，通过watch来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理。

## 14、Nginx作用以及常见配置

作用：反向代理、负载均衡、动静分离（静态资源服务器，可以进行图片等资源管理）

配置：

配置文件默认为安装目录下的conf/nginx.conf，如果有使用到其他子配置文件，可以在nginx.conf中使用include 文件路径;的方式加载使用，比如server段，就可以单独写成一个配置文件，在http段下面使用include加载使用。

nginx.conf配置的结构

全局配置区域

...

http {

...

http段配置

...

server {

......

......

location {

...

...

}

}

server {

......

}

}

nginx.conf配置文件的常见配置参数：

nginx的工作进程运行时的身份，也就是进程文件的属主和属组属性，如果在源码安装时configure配置已经指定用户和组，这里可以注释掉

#user nobody;

# 定义nginx的工作进程的数量，一般为CPU核数或核数的倍数，该参数与并发量有关

worker\_processes 1;

# 错误日志的位置

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

# nginx的master进程的pid存储文件，默认为/usr/local/nginx/logs/nginx.pid

#pid logs/nginx.pid;

events {

# 每一个工作进程可以接收的请求连接数，一般与系统的进程可以打开的文件描述符数量相同,

worker\_connections 1024; # 并发量=工作进程数量\*连接数

}

http {

# mime.types文件含有nginx支持的媒体类型，include可以加载该文件

include mime.types;

default\_type application/octet-stream;

# 访问日志的格式，可以自定义

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

# '$status $body\_bytes\_sent "$http\_referer" '

# '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

# 指定访问日志的位置和格式main

#access\_log logs/access.log main;

# 自定义日志格式，名称为shoplog，便于后续使用

log\_format shoplog '$remote\_addr@$time\_local@$status@$http\_user\_agent';

# 调用系统的方法传输文件，速度更快，

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65; # 客户端连接的超市时间，单位为秒，65s后自动断开

# 隐藏nginx版本号，不再浏览显示

server\_tokens off;

#gzip on;

server {

listen 80;

server\_name 192.168.211.109;

# 根目录，浏览器访问时默认在该目录下寻找资源文件

root html/web;

# 开启列表展示功能

#autoindex on;

# 人性化方式显示文件的大小

#autoindex\_exact\_size off;

# 显示本地时间

#autoindex\_localtime on;

# fancy-index第三方模块提供的功能，使文件列表展示的更优美

#fancyindex on;

#fancyindex\_exact\_size off; # 人性化方式显示文件的大小

location / {

# 执行默认寻找index.html

index index.html;

# 反向代理，将请求交给http://192.168.211.101:80服务器处理

# proxy\_pass http://192.168.211.101:80;

}

#location ~\* /1.HTML {

# error\_page 404 = @html\_err;

#}

#location @html\_err {

# return 501;

#}

}

server {

listen 80;

server\_name score.devops.com;

# 域名重定向

rewrite / http://shop.devops.com permanent;

}

server {

listen 80;

server\_name shop.devops.com;

root html/tp5shop/public;

# gzip on压缩功能，将服务器传输的文件压缩返回给浏览器，可以减少传输的数据量，提供性能，一般浏览器是支持解压的

gzip on; #开启压缩功能

gzip\_http\_version 1.0; # 指定http的版本

gzip\_disable 'MSIE [1-6]'; # 禁止IE的1~6版本使用该功能

gzip\_types application/javascript text/css image/jpeg image/png; # 指定压缩哪些类型的文件

# 禁止ip访问，当有匹配时，就不会在向下匹配

# deny all; # 拒绝所有

# allow 192.168.211.1; # 允许192.168.211.1

# 用户访问限制

# auth\_basic 'pls login:'; # 指定提示语"pls login:"

# auth\_basic\_user\_file /usr/local/nginx/conf/userlist; # 指定授权用户所在文件

# 基于域名的日志分割，所有访问shop.devops.com域名的访问日志记录在该文件中

access\_log /usr/local/nginx/logs/shop.devops.com shoplog;

location / {

# expires 设置客户端缓存

#expires 1h;

index index.php index.html;

# 资源重定向，如访问http://shop.devops.com/index.html后会被重写为访问http://shop.devops.com/index.php，permanent表示永久重定向

rewrite /index.html /index.php permanent;

# 资源重定向，$request\_filename为nginx的内置变量，表示资源文件路径

if (!-e $request\_filename) {

rewrite ^(.\*)$ /index.php?s=/$1 last;

break;

}

}

# 资源重定向

#location /index {

# rewrite /index.html /index.php last;

#}

location ~ \.(js|css|jpg|png) {

# 告诉客户端所有js,css,jpg,png文件都可以缓存1小时，不用重新在服务器下载

expires 1h;

# 防盗链实现，所有不是从shop.devops.com跳转过去访问js|css|jpg|png文件的都被拦截，返回404

valid\_referers shop.devops.com;

if ($invalid\_referer) {

return 404;

}

}

# php解析

location ~ \.php$ {

# root html;

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

}

}

# 第十章 Git与Linux篇

## 1、reset 与 rebase, pull 与 fetch 的区别

git reset 不修改commit相关的东西，只会去修改.git目录下的东西。

git rebase 会试图修改你已经commit的东西，比如覆盖commit的历史等，但是不能使用rebase来修改已经push过的内容，容易出现兼容性问题。rebase还可以来解决内容的冲突，解决两个人修改了同一份内容，然后失败的问题。

git pull pull=fetch+merge,

使用git fetch是取回远端更新，不会对本地执行merge操作，不会去动你的本地的内容。 pull会更新你本地代码到服务器上对应分支的最新版本

## 2、git merge和git rebase的区别

git merge把本地代码和已经取得的远程仓库代码合并。

git rebase是复位基底的意思，gitmerge会生成一个新的节点，之前的提交会分开显示，而rebase操作不会生成新的操作，将两个分支融合成一个线性的提交。

## 3、git如何解决代码冲突

第一种：

git stash

git pull

git stash pop

这个操作就是把自己修改的代码隐藏，然后把远程仓库的代码拉下来，然后把自己隐藏的修改的代码释放出来，让gie自动合并。

如果要代码库的文件完全覆盖本地版本。

git reset –hard

git pull

第二种：通过开发工具 idea进行merge代码合并

## 4、Linux常用命令

|  |  |  |
| --- | --- | --- |
| 序号 | 命令 | 命令解释 |
| 1 | top | 查看内存 |
| 2 | df -h | 查看磁盘存储情况 |
| 3 | iotop | 查看磁盘IO读写(yum install iotop安装） |
| 4 | iotop -o | 直接查看比较高的磁盘读写程序 |
| 5 | netstat -tunlp | grep 端口号 | 查看端口占用情况 |
| 6 | uptime | 查看报告系统运行时长及平均负载 |
| 7 | ps aux | 查看进程 |

## 5、如何查看测试项目的日志

一般测试的项目里面，有个logs的目录文件，会存放日志文件，有个xxx.out的文件，可以用tail -f 动态实时查看后端日志

先cd 到logs目录(里面有xx.out文件)

>tail -f xx.out

这时屏幕上会动态实时显示当前的日志，ctr+c停止

## 6、如何查看最近1000行日志

>tail -1000 xx.out

## 7、LINUX中如何查看某个端口是否被占用

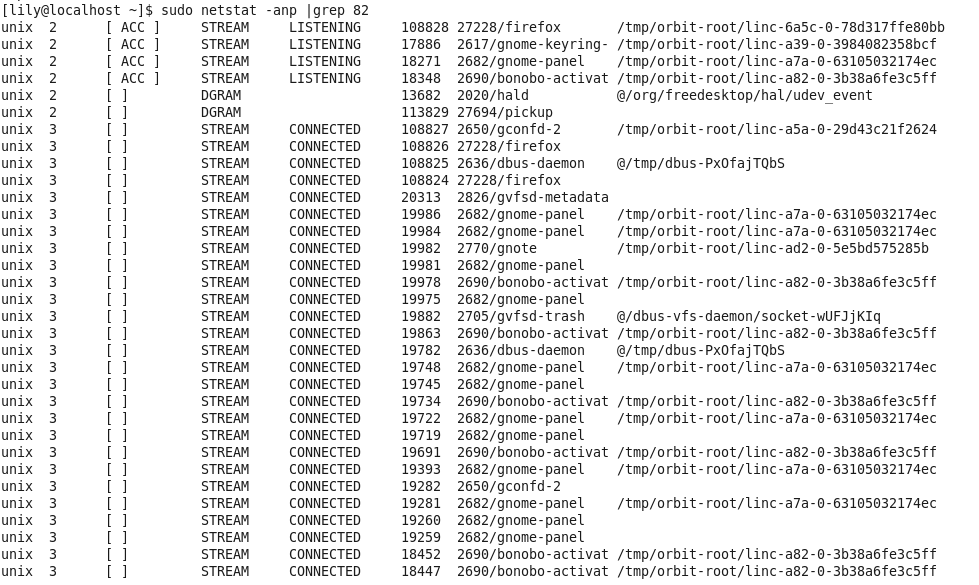
>netstat -anp | grep 端口号

IMG_256

图中主要看监控状态为LISTEN表示已经被占用，最后一列显示被服务mysqld占用，查看具体端口号，只要有如图这一行就表示被占用了

查看82端口的使用情况，如图

>netstat -anp |grep 82

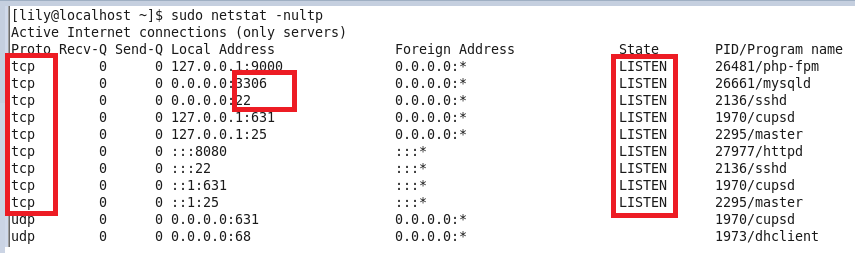


可以看出并没有LISTEN那一行，所以就表示没有被占用。此处注意，图中显示的LISTENING并不表示端口被占用，不要和LISTEN混淆哦，查看具体端口时候，必须要看到tcp，端口号，LISTEN那一行，才表示端口被占用了

## 8、查看当前所有已经使用的端口情况

如图：

netstat -nultp（此处不用加端口号）



# 第十一章 电商项目篇之尚品汇商城

## 一、常见共性问题

## 1、介绍下最近做的项目

可以从2个方向出发

介绍项目背景、项目功能和自己负责的功能模块

介绍项目背景、项目使用技术栈和自己负责的功能模块

### 1.1 项目背景：

可以介绍项目是什么类型的（B2C、B2B2C、O2O这类），为什么要做这个项目，

有工作经验的找工作一般这样介绍：项目是自己公司开发，自己运营的，然后不断加功能进行迭代和维护；或者是项目定制的，给甲方客户开发的一个项目，上线后不负责维护和迭代，这样避免了很多后期问题，这两种都可以。

咱们可以借鉴以上的方式去介绍，刨除公司情况，以学习为主。

### 1.2 项目功能：

结合项目，进行主要的功能模块阐述，可以结合电商项目的核心购物流程去说：后台管理系统（商品的管理）、商品详情、商品搜索、购物车、单点登录+社交登录、订单、支付、秒杀等等。

### 1.3 技术栈：

使用springboot整合SpringCloud 以及MyBatis-Plus进行微服务构建，使用nacos作为注册中心和配置中心，使用feign进行服务远程调用，使用gateway网关进行请求负载、请求过滤、统一鉴权和限流，使用Sentinel进行服务的熔断和降级，使用Spring Cloud Sleuth进行链路追踪，针对于项目图片文件资源较多，采用FastDFS进行文件资源存储，使用redis数据库进行数据缓存以及分布式锁的实现，使用ElasticSearch进行商品的搜索业务实现…..（这块基础架构说完后，主要结合自己负责的功能模块去说技术点的应用）

### 1.4 自己负责的功能模块：

已简历为主，简历上写了哪几个，就说那几个，一定要知道自己简历写的什么内容。

### 1.5 项目介绍参考：

尚品汇商城是B2C模式的综合性在线销售平台。商城分为后台管理部分与用户前台使用部分。后台管理部分包括：商品管理模块（商品分类、品牌、平台属性、SPU与SKU以及销售属性、商品上下架和商品评论管理等）、内容广告模块、库存管理模块、订单管理模块、促销管理（秒杀等商品设置）、客户模块、统计报表模块和系统基础权限等模块。

用户前台使用部分：商城首页、商品搜索（可按条件查询展示）、商品详情信息展示、购物车、用户单点登录和社交登录（微信登录）、用户会员中心、订单的创建修改、展示以及在线支付（支付宝、微信）、物流模块、商品评论以及秒杀活动等功能。

### 1.6 项目架构图：



### 1.7 整体业务介绍：

|  |  |
| --- | --- |
| 首页 | 静态页面，包含了商品分类，搜索栏，商品广告位。 |
| 全文搜索 | 通过搜索栏填入的关键字进行搜索，并列表展示 |
| 分类查询 | 根据首页的商品类目进行查询 |
| 商品详情 | 商品的详细信息展示 |
| 购物车 | 将有购买意向的商品临时存放的地方 |
| 单点登录 | 用户统一登录的管理 |
| 结算 | 将购物车中勾选的商品初始化成要填写的订单 |
| 下单 | 填好的订单提交 |
| 支付服务 | 下单后，用户点击支付，负责对接第三方支付系统。 |
| 订单服务 | 负责确认订单是否付款成功，并对接仓储物流系统。 |
| 仓储物流 | 独立的管理系统，负责商品的库存。 |
| 后台管理 | 主要维护类目、商品、库存单元、广告位等信息。 |
| 秒杀 | 秒杀抢购完整方案 |

### 1.8 后台管理系统功能：

电子商务网站整个系统的后端管理，按功能划分为九大模块，包括商品组织管理、订单处理、内容发布管理等模块。

#### 1.8.1 后台主页：

各类主要信息的概要统计，包括客户信息、 订单信息、商品信息、库存信息、评论和最近反馈等。

#### 1.8.2 商品模块：

##### 1).商品管理：

商品SPU和SKU的添加、修改、 删除、复制、批处理、商品计划上下架、SEO、商品多媒体上传等，可以定义商品是实体还是虚拟，可以定义是否预订、是否缺货销售等。

##### 2).商品分类管理：

树形的商品目录组织管理，并可以设置品类关联与商品推荐。

##### 3).商品平台属性管理：

定义商品的属性类型，设置自定义属性项。

##### 4).品牌管理：

添加、修改、删除、上传品牌 LOGO。

##### 5).商品评论管理：

商品评论的搜索、条件查询列表展示、回复、删除等功能。

#### 1.8.3 销售模块：

##### 1).促销秒杀管理：

设置秒杀商品、购物车促销和 优惠券促销三类，可以随意定义不同的促销规则，满足日常促销活动：购物折扣、购物赠送积分、购物赠送优惠券、购物免运输费、特价商品、特定会员购买特定商品、折上折、买二送一等。

##### 2).礼券、积分管理：

比如添加、发送礼券和积分

##### 3).关联/推荐管理：

基于规则引擎，可以支持多种推荐类型，可手工添加或者自动评估商品。

#### 1.8.4 订单模块：

##### 1).订单管理：

可以编辑、解锁、取消订单、 拆分订单、添加商品、移除商品、确认可备货等，也可对因促销规则发生变化引起的价格变化进行调整。订单处理完可发起退货、换货流程。

##### 2).支付：

常用于订单支付信息的查看和手工 支付两种功能。手工支付订单，常用于“款到发货”类型的订单，可理解为对款到发货这类订单的一种补登行为。

##### 3).结算：

提供商家与第三方物流公司的结算 功能，通常是月结。同时，结算功能也是常用来对“货到付款”这一类型订单支付后的数据进行对帐

#### 1.8.5 库存模块：

##### 1).库存管理：

引入库存的概念，不包括销售规则为永远可售的商品，一个SKU对应一个库存量。库存管理提供增加、减少等调整库存量的功能;另外，也可对具具体的SKU设置商品的保留数量、最小库存量、再进货数量。每条SKU商品的具体库存操作都会记录在库存明细记录里边。

##### 2).查看库存明细记录。

##### 3).备货/发货：

创建备货单、打印备货单、打印发货单、打印快递单、完成发货等一系列物流配送的操作。

##### 4).退/换货：

对退/换货的订单进行收货流程的处理。

#### 1.8.6 内容模块：

##### 1).内容管理：

包括内容管理以及内容目录管理。内容目录由树形结构组织管理。类似于商品目录的树形结构，可设置目录是否为链接目录。

##### 2).广告管理：

添加、修改、删除、上传广告、 定义广告有效时限。

##### 3).可自由设置商城导航栏目以及栏目内容、栏目链接。

#### 1.8.7 客户模块：

##### 1).客户管理：

添加、删除、修改、重设密码、 发送邮件等。  

##### 2).反馈管理：

删除、回复。

##### 3).消息订阅管理：

添加、删除、修改消息组 和消息、分配消息组、查看订阅人。

##### 4).会员资格：

添加、删除、修改。

#### 1.8.8 系统模块：

##### 1).安全管理：

管理员、角色权限分配和安全日志

##### 2).系统属性管理：

用于管理自定义属性。可关联模块包括商品管理、商品目录管理、内容管理、客户管理。

##### 3).运输与区域：

运输公司、运输方式、运输 地区。

##### 4).支付管理：

支付方式、支付历史。

##### 5).包装管理：

添加、修改、删除。

##### 6).数据导入管理：

商品目录导入、商品导入、 会员资料导入。

#### 1.8.9 报表模块：

   缺省数个统计报表，支持时间段过滤、支持按不同状态过滤、支持HTML、PDF和Excel格式的导出和打印。

1.用户注册统计   2.低库存汇总   3.缺货订单   4.订单汇总   5.退换货

## 2、项目开发周期：

**开发、维护和运营一体化的：**开发周期8个月左右，后期维护与迭代时间会更长

**项目定制：**前期架构+数据库设计+编码+开发+测试解bug共7个月左右进行项目交付

**培训学习项目:** 20天教程（咱们是学习20天课程）

## 3、项目参与人数：

**一般公司:**项目经理（PM）1人、产品（PD）2人、界面设计（UI）2人、前端 3人、Java后台（DE）6人，其中1人是开发组长、测试（QA）2人、运维（SRE）1人

**培训学习项目：**根据课程内容编写代码，自己实现部分功能

## 4、公司开发相关各岗位职责：

### **4.1 项目经理（PM）：**

企业建立以项目经理责任制为核心，对项目实行质量、安全、进度、成本管理的责任保证体系和全面提高项目管理水平设立的重要管理岗位。职责：

1、负责软件项目管理及计划实施；

2、具备较强管理、协调及沟通能力，帮助开发人员解决开发过程中遇到的技术问题，做好日常的开发团队管理工作；

3、与各团队协同工作，确保开发工作正常顺利的开展；

### 4.2 产品（PD）：

企业中专门负责产品管理的职位，负责调查并根据用户的需求，确定开发何种产品，选择何种技术、商业模式等。并推动相应产品的开发组织，他还要根据产品的生命周期，协调研发、营销、运营等，确定和组织实施相应的产品策略，以及其他一系列相关的产品管理活动。职责： 1. 根据公司产品及用户需求，结合市场调研情况，进行产品规划；

2. 负责用户沟通、需求分析诊断；

3. 负责产品定位、用户体验流程定位及产品设计；

4. 推动、协调与控制产品策划及研发工作，保证产品需求的有效实现；

5. 负责产品持续升级，不断提升用户满意度及忠诚度；

6. 对行业及竞争产品的分析，跟踪最新发展趋势，并提交分析报告。

### 4.3 界面设计（UI）：

对软件的人机交互、操作逻辑、界面美观的整体设计。职责：

1、负责公司产品PC端和移动端的UI界面设计工作；

2、配合完成校样修改和界面调整；

3、深入了解负责的产品，并通过各种设计形式和视觉语言让用户感受到产品的优点和特性；

4、跟进设计的变化和需求，注重相关文档的整理、资料的收集；能独立完成界面设计工作。

### 4.4 开发组长（TL）：

其实就是个更小一点的项目经理。其职责：1、 参与软件的设计负责系统需求的分析，进行系统设计和数据库设计；

2、 解决开发过程中技术问题和提供解决办法；

3、 能够带领小组负责模块的功能开发；

4、 负责项目组代码的审查工作，有效地控制项目的质量风险。

### 4.5 测试（QA）：

测试工程师，软件质量的把关者，工作起点高，发展空间大。职责： 1.理解、分析需求文档，挖掘、细化需求;

2.根据软件需求及设计文档编写测试用例，参与文档评审并维护相关文档;

3.准备测试数据，执行测试用例，记录测试结果，整理测试报告;

4.负责BUG的提交、跟踪、验证、关闭;

5.负责测试部门测试环境及BUG系统管理与维护。

6.对产品进行必要的功能，性能，安全，兼容性及其它方面的测试工作；

7.公司安排的其它工作。

### 4.5 运维（SRE）：

运维工程师最基本的职责都是负责服务的稳定性。

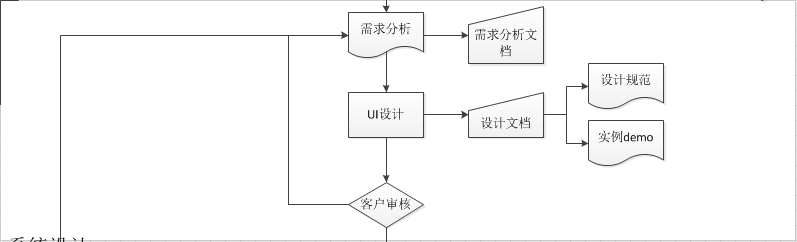
1. 产品发布前：负责参与并审核架构设计的合理性和可运维性，以确保在产品发布之后能高效稳定的运行。

2. 产品发布阶段：负责用自动化的技术或者平台确保产品可以高效的发布上线，之后可以快速稳定迭代。

3. 产品运行维护阶段：负责保障产品7\*24H稳定运行，在此期间对出现的各种问题可以快速定位并解决；在日常工作中不断优化系统架构和部署的合理性，以提升系统服务的稳定性。

## 5、企业项目开发流程:

### 3.1 需求分析



项目前期主要指的是项目业务需求调研、包括配合用户制定项目建设方案、技术规范书、配合市场人员进行售前技术交流等环节，此阶段应该组织由售前工程师、需求分析师以及系项目经理等组成一个临时小组，负责跟踪项目。这个小组根据项目的大小和客户的要求确定小组成员。

项目前期小组的工作是项目的开始，这个小组工作成绩的优劣、工作质量的高低，将直接影响项目的成败。因此，从管理层的角度，一定要重视这个环节。

项目前期小组需要完成的工作包括以下方面：

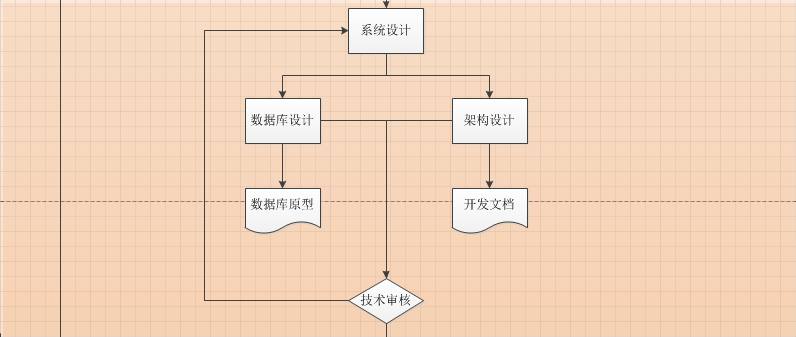
1、 客户的各种项目前期要求，如：方案介绍、业务需求编写等

2、 提交项目可行性分析报告，包括成本/效益分析

3、 提交项目建议方案

4、 提交业务需求说明书或需求分析说明书

### 3.2 系统设计



系统设计是决定项目或软件系统“怎样做”的过程，这个过程回答了系统应该如何实现的问题。从软件工程的角度，设计阶段大约是整个项目开发成本的25%，所以，设计团队以及该团队的工作成绩对于整个系统来说至关重要。

人员需求

设计团队一般由3—8名设计人员组成，从这个阶段起，项目需要一名项目经理，行使项目组的各种管理职能。设计团队的成员具体包括：

1名项目经理

包括1—2名项目前期成员

1名系统构架师

1名数据库设计人员

1名用户界面设计人员组成

设计团队需要完成的工作包括：

1、项目开发计划

2、确定系统软硬件配置最佳方案

3、确定系统开发平台以及开发工具

4、确定系统软件结构

5、确定系统功能模块以及各个模块之间的关系

6、确定系统测试方案

7、提交系统数据库设计方案

8、提交系统概要设计文档

由于应用软件需求经常变化，因此设计需要考虑系统可扩展性，并需要在设计过程中对于重要的环节和用户进行及时沟通。

### 3.3 编码开发



将用户的需求变成真正可用的软件系统，是通过编码和系统实现阶段来完成的。虽然软件的质量主要取决于系统设计的质量，但是编码的途径和实现的具体方法对程序的可靠性、可读性、可测试性和可维护性产生深远的影响。

人员需求

这个阶段要根据用户对项目进度的要求灵活组织开发团队。为了工作的连贯性，同时也为了解决在开发过程中用户需求有可能变化的因素，开发团队应该保留1—3名设计团队的成员。

人员分工

开发过程中，项目经理的角色非常重要，项目经理负责项目组开发人员的日常管理，控制项目的进度，负责和设计部门、市场部门以及客户之间进行必要的沟通。这个阶段通常是多个部门的人员共同组成一个项目组，因此，项目管理的一定要保证统一管理，理想状态是项目经理全权负责项目组人员的人员工作安排、业绩考核、工资奖金等，因为项目经理最了解项目组成员的工作态度和工作业绩。

一般在大型项目开发团队中，应该设立专门的技术经理岗位，负责对项目组的技术方案进行管控，技术经理最好是由设计团队中抽调出来。技术经理在项目开发过程中需要注意程序风格、编码规范等问题，并必须进行有效的代码管理（版本管理）。

开发过程还应该进行系统的单元测试工作，确保各个独立模块功能的正确性和性能满足需求说明书的要求。

开发团队应该完成的工作包括：

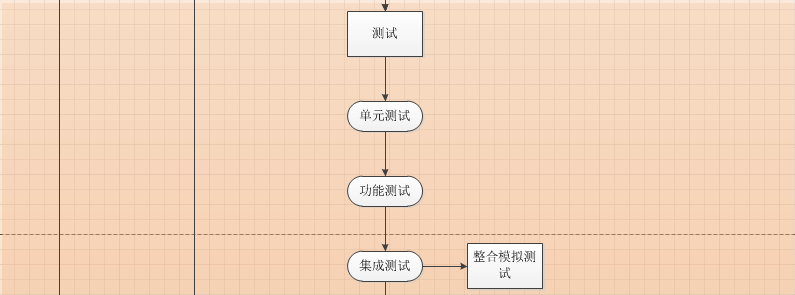
1、 系统的实现代码编写

2、 单元测试

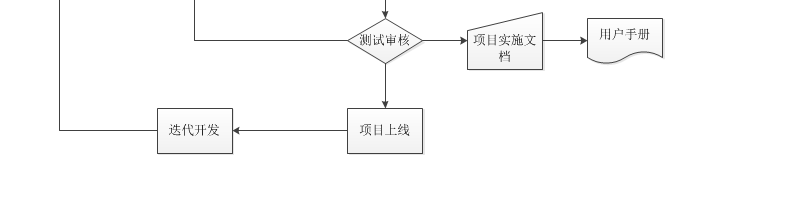
3、 提交源代码清单

4、 提交单元测试报告

### 3.4 系统测试



### 3.5 部署实施



由于从事的应用软件的开发，因此，在开发完成之后经常会有系统集成、软件的安装等工作。这个阶段还经常伴随着新的业务需求和本地化需求的产生，因此将会有一部分的开发工作需要在这个阶段完成。

## 6、项目版本控制：

使用git进行版本控制，剩下的主要是对git仓库的一些列操作，要记住操作命令，以及分支切换等等。

## 7、一般项目服务器数量：

### 开发测试阶段：

开发在自己电脑上开发，代码提交到git仓库（gitlab、gitee等）的开发分支，到测试时，公司有1-2台测试服务器，把开发分支代码合并到测试分支，使用jenkins进行测试环境代码部署，测试人员进行测试。

### 生产环境：

为了保证高可用，首先每个服务都要进行 集群部署，包括项目中依赖的第三方服务，这样的话，服务器的数量是很庞大的，以尚品汇商城所学功能实现为例，如下：

Nginx2台（主备）；Nacos 3台（官方推荐最少3台）；项目中有服务数量\*2（19\*2=38）共需要38台；redis无中心化集群6台（3主3从），如果是主从哨兵集群5台；mysql数据库（读写分离的情况下，1主2从 4\*3 =12）共12台；ES集群3台；rabbitmq集群2台；fastDFS集群（保证高可用情况下，traker2台，storage2组4台）共6台。

以上所述 服务器数量一共为72台，数量很多，以目前市场硬件服务器平均一台5万块钱来计算，72\*5=360万，投入成本和维护成本很大，在项目前期没有大用户量的情况下，不建议。

针对于这个问题，如果是项目定制（外包）的话，可以由甲方（客户）自己去部署，这种情况也是存在的。

还有一种情况是在项目前期没有那么大的用户量情况下，可以采用单机（主备）部署方案，把所有服务部署同一台服务器上，进行资源节省。

## 8、上线后QPS，用户量、同时在线人数并发数等问题：

建议采用项目定制方案：项目给客户做的，甲方的数据我们拿不到。

但是要了解以下数据关键词：

（1）QPS（TPS）：每秒钟request/事务 数量

（2）并发数： 系统同时处理的request/事务数

（3）响应时间： 一般取平均响应时间

QPS（TPS）= 并发数/平均响应时间或者并发数 = QPS\*平均响应时间 一个典型的上班签到系统，早上8点上班，7点半到8点的30分钟的时间里用户会登录签到系统进行签到。公司员工为1000人，平均每个员上登录签到系统的时长为5分钟。可以用下面的方法计算。 QPS = 1000/(30\*60) 事务/秒 平均响应时间为 = 5\*60 秒 并发数= QPS\*平均响应时间 = 1000/(30\*60) \*(5\*60)=166.7

说明：

一个系统吞吐量通常由QPS（TPS）、并发数两个因素决定，每套系统这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。

我们做项目要排计划，可以多人同时并发做多项任务，也可以一个人或者多个人串行工作，始终会有一条关键路径，这条路径就是项目的工期。系统一次调用的响应时间跟项目计划一样，也有一条关键路径，这个关键路径是就是系统影响时间； 关键路径是有CPU运算、IO、外部系统响应等等组成。

## 9、如何解决并发问题的？

尚品汇商城是微服务架构构建，集群部署，使用nginx做负载，数据库的读写分离，redis缓存，页面静态化等这些都是解决并发的手段。

## 10、你做完这个项目后有什么收获？

首先，在数据库方面，我现在是真正地体会到数据库的设计真的是一个程序或软件设计的重要和根基。因为数据库怎么设计，直接影响到一个程序或软件的功能的实现方法、性能和维护。由于我做的模块是要对数据库的数据进行计算和操作的，所以我对数据库的设计对程序的影响是深有体会，就是因为我们的数据库设计得不好，搞得我在对数据库中的数据进行获取和计算利润、总金时，非常困难，而且运行效率低，时间和空间的复杂也高，而且维护起来很困难，过了不久，即使自己有注释，但是也要认真地看自己的代码才能明白自己当初的想法和做法。加上师兄的解说，让我对数据库的重要的认识更深一层，数据库的设计真的是重中之重。

其次，就是分工的问题。虽然这次的项目我们没有在四人选出一个组长，但是，由于我跟其他人都比较熟，也有他们的号码，然后我就像一个小组长一样，也是我对他们进行了分工。俗话也说，分工合作，分好了工，才能合作。但是这次项目，我们的分工却非常糟糕，我们在分工之前分好了模块，每个 责哪些模块。本以为我们的分工是明确的，后来才发现，我们的分工是那么的一踏糊涂，一些功能上紧密相连的模块分给了两个人来完成，使两个人都感到迷惘，不知道自己要做什么，因为两个人做的东西差不多。我做的，他也在做，那我是否要继续做下去？总是有这样的疑问。从而导致了重复工作，浪费时间和精力，并打击了队员的激情，因为自己辛辛苦苦写的代码，最后可能没有派上用场。我也知道，没有一点经验的我犯这样的错是在所难免，我也不过多地怪责自己，吸取这次的教训就好。分工也是一门学问。 再者，就是命名规范的问题。可能我们以前都是自己一个人在写代码，写的代码都是给自己看的，所以我们都没有注意到这个问题。就像师兄说的那样，我们的代码看上去很上难看很不舒服，也不知道我们的变量是什么类型的，也不知道是要来做什么的。但是我觉得我们这一组人的代码都写得比较好看，每个人的代码都有注释和分隔，就是没有一个统一的规范，每个人都人自己的一个命名规则和习惯，也不能见名知义。还有就是没有定义好一些公共的部分，使每个人都有一个自己的“公共部分”，从而在拼起来时，第一件事，就是改名字。而这些都应该是在项目一开始，还没开始写代码时应该做的。 然后，我自己在计算时，竟然太大意算错了利润，这不能只一句我不小心就敷衍过去，也是我的责任，而且这也是我们的项目的核心部分，以后在做完一个模块后，一定要测试多次，不能过于随便地用一个数据测试一下，能成功就算了，要用可能出现的所有情况去测试程序，让所有的代码都有运行过一次，确认无误。

最后，也是我比较喜欢的东西，就是大家一起为了一个问题去讨论和去交流。因为我觉得，无论是谁，他能想的东西都是有限的，别人总会想到一些自己想不到的地方。跟他人讨论和交流能知道别人的想法、了解别人是怎样想一个问题的，对于同样的问题自己又是怎样想的，是别人的想法好，还是自己的想法好，好在什么地方。因为我发现问题的能力比较欠缺，所以我也总是喜欢别人问我问题，也喜欢跟别人去讨论一个问题，因为他们帮我发现了我自己没有发现的问题。在这次项目中，我跟植荣的讨论就最多了，很多时候都是不可开交的那种，不过我觉得他总是能够想到很多我想不到的东西，他想的东西也比我深入很多，虽然很多时候我们好像闹得很僵，但是我们还是很要好的! 嘻嘻！而且在以后的学习和做项目的过程中，我们遇到的问题可能会多很多，复杂很多，我们一个人也不能解决，或者是没有想法，但是懂得与他人讨论与交流就不怕这个问题，总有人的想法会给我们带来一片新天地。相信我能做得更好。 还有就是做项目时要抓准客户的要求，不要自以为是，自己觉得这样好，那样好就把客户的需求改变，项目就是项目，就要根据客户的要求来完成。

## 11、在做这个项目的时候你碰到了哪些问题？你是怎么解决的？

（1）开发SpringBoot接口出现客户端和服务端不同步，导致接口无法测试，产生的原因沟通不畅。

（2）订单提交时由于本地bug或者意外故障导致用户钱支付了但是订单不成功，采用对账方式来解决。

（3）上线的时候一定要把支付的假接口换成真接口。

（4）项目中用到了曾经没有用过的技术，解决方式：用自己的私人时间主动学习

（5）在开发过程中与测试人员产生一些问题，本地环境ok但是测试环境有问题，环境的问题产生的，浏览器环境差异，服务器之间的差异

（6）系统运行环境问题，有些问题是在开发环境下OK，但是到了测试环境就问题，比如说系统文件路径问题、导出报表中的中文问题（报表采用POI），需要在系统jdk中添加相应的中文字体才能解决；

## 二、介绍功能模块

这块面试官的问法不同，有的会让你从你负责的模块中，挑选一个你认为做的最好的或者有亮点的模块去说，有的面试官是把你负责的模块问个遍，所以只要你简历上写了是你负责的模块，一定要去弄的非常熟练，如果你自己做的东西，你都不知道，那对于整个项目来说也就没必要再问了，所以这块是重点。在介绍自己负责的功能模块时，可以通过2个方向去介绍，一个是模块业务，另一个是模块开发中的技术解决方案。这两个方向全掌握了，才是真正的把负责的模块掌握了。

下面是所学项目中所有功能模块的业务以及常见面试问题，仅供参考，不要去背，要结合项目进行理解为主。

## 1、商品管理模块

数据库表设计：（重点，要知道表中字段与表直接的关系）



### 1.1 业务参考话术：

我们这个电商平台是不支持商家入驻功能的，因此整个商品管理主要是我们这个电商后台系统的一个核心功能模块。主要的功能分为：

对于商品品牌、商品分类、销售属性和平台属性进行管理的一些操作；

对商品有对应操作权限的业务员对于商品的添加、上下架、修改、删除、批量操作、模糊查询等功能。

针对商品这个模块来说，因为不同的分类对应不同的平台属性，平台属性和SKU进行关联，用于后期商品搜索功能实现，可以通过平台属性查询出所选属性下的所有SKU信息；

同时，一个分类下又有多个品牌，品牌下对应多个SPU商品，每一个SPU下有多种销售属性，我们通过商品SPU的不同销售属性组合生成多个SKU。例如 iPhone7是一个SPU，由于iPhone7分内存、颜色等销售属性，内存有64G、128G和256G，颜色有红色、金色和黑色，这样每种不同销售属性组合就是一个新的SKU，64G+黑色、64G+金色、64G+红色，128G+金色…….等。（主要就是说表设计和表关系）

### 1.2 商品添加

先判断是否有对应的商品添加的权限，如果没有，直接不显示操作按钮；

如果有，先选择商品所属的分类，然后商品分类选择之后会对应的查询出所有平台属性和所关联的所有品牌；

然后添加产品SPU的一些基本信息（SPU名称、标题、商城价格等）；

其次，上传产品SPU对应的图片，这块因为整个电商平台中需要保存大量的图片，因此我们使用了一个分布式文件存储系统FastDFS来存储商品的图片，这样的话后期也可针对容量进行水平扩展，不影响原来的使用，并且针对于高并发、高可用的问题，FastDFS的容灾性、负载均衡也是个优势；

在SPU列表中有添加sku的按钮，点击后可以添加给SPU添加SKU。进入SKU添加页面时会查出当前SPU的所有销售属性和图片，然后添加SKU信息，选择对应的销售属性，选择对应的图片。调用后台的商品添加接口/服务，然后对应的接口/服务中生成商品添加时的一些默认数据（添加时间、更新时间、操作人等），在商品添加的时候我们默认是下架状态。需要拥有审核功能的业务员针对商品中是否有一些不合法的数据进行审核。

### 1.3 商品上下架

因为商品添加之后，默认是下架状态，如果商品想处于销售状态，必须将商品的状态改为上架状态。在SKU列表页面中有上、下架操作按钮：

上架时将SKU信息添加Redis进行缓存，将SKU信息添加到ES一份以供搜索

修改上架状态 0未上架 1已上架 2已下架

### 1.4 模块相关问题：

#### 1）SPU与SKU的概念，SKU是怎么划分的？

比如，咱们购买一台iPhoneX手机，iPhoneX手机就是一个SPU，但是你购买的时候，不可能是以iPhoneX手机为单位买的，商家也不可能以iPhoneX为单位记录库存。必须要以什么颜色什么版本的iPhoneX为单位。比如，你购买的是一台银色、128G内存的、支持联通网络的iPhoneX ，商家也会以这个单位来记录库存数。那这个更细致的单位就叫库存单元（SKU）。

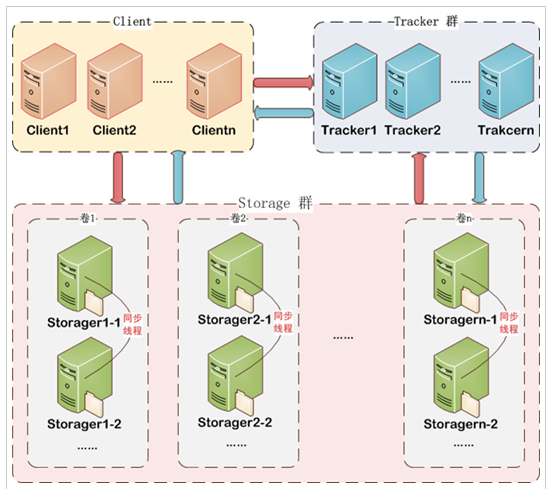
#### 2）介绍下FastDFS？

开源的分布式文件系统，主要对文件进行存储、同步、上传、下载，有自己的容灾备份、负载均衡、线性扩容机制；

FastDFS架构主要包含Tracker server和Storage server。客户端请求Tracker server进行文件上传、下载的时候，通过Tracker server调度最终由Storage server完成文件上传和下载。

Tracker server：跟踪器或者调度器，主要起负载均衡和调度作用。通过Tracker server在文件上传时可以根据一些策略找到Storage server提供文件上传服务。

Storage server：存储服务器，作用主要是文件存储，完成文件管理的所有功能。客户端上传的文件主要保存在Storage server上，Storage server没有实现自己的文件系统而是利用操作系统的文件系统去管理文件。



存储服务器采用了分组/分卷的组织方式。整个系统由一个组或者多个组组成；

组与组之间的文件是相互独立的；所有组的文件容量累加就是整个存储系统的文件容量；一个组可以由多台存储服务器组成，一个组下的存储服务器中的文件都是相同的，组中的多台存储服务器起到了冗余备份和负载均衡的作用；在组内增加服务器时，如果需要同步数据，则由系统本身完成，同步完成之后，系统自动将新增的服务器切换到线上提供使用；当存储空间不足或者耗尽时，可以动态的添加组。只需要增加一台服务器，并为他们配置一个新的组，即扩大了存储系统的容量。

## 2、商品详情模块

### 2.1 业务参考话术：

商品详情页，就是以购物者的角度展现一个sku的详情信息。这个页面不同于传统的页面，使用者并不是管理员，需要对信息进行查删改查，取而代之的是点击购买、放入购物车、切换颜色等等。另外一个特点就是该页面的高访问量，虽然只是一个查询操作，但是由于频繁的访问所以我们必须对其性能进行最大程度的优化。所以我们页面数据放入Redis缓存来提高性能，为了防止缓存击穿，我们采用了分布式锁。由于商品详情展示的数据需要要调用多个查询接口，为了提高响应速度，我们采用线程异步编排的方式进行接口调用。

### 2.2 模块相关问题：

Redis是这块的重点之一，有关reids的高频问题，详见高频面试题，这里只说项目相关redis常见问题

#### 1）Redis在你们项目中是怎么用的？

商品详情中的数据放入缓存，价格是实时获取的；

单点登录系统中也用到了redis。因为我们是微服务系统，把用户信息存到redis中便于多系统之间获取共用数据；

我们项目中同时也将购物车的信息设计存储在redis中，用户未登录采用UUID作为Key，value是购物车对象；用户登录之后将商品添加到购物车后存储到redis中，key是用户id，value是购物车对象；

因为针对评论这块，我们需要一个商品对应多个用户评论，并且按照时间顺序显示评论，为了提高查询效率，因此我们选择了redis的list类型将商品评论放在缓存中；

在统计模块中，我们有个功能是做商品销售的排行榜，因此选择redis的zset结构来实现；

#### 2）缓存击穿解决（分布式锁）

缓存击穿是指对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个key在大量请求同时进来之前正好失效，那么所有对这个key的数据查询都落到db，我们称为缓存击穿。

使用分布式锁，采用redis的KEY过期时间实现

Redis:命令

# set skuid:1:info “OK” NX PX 10000

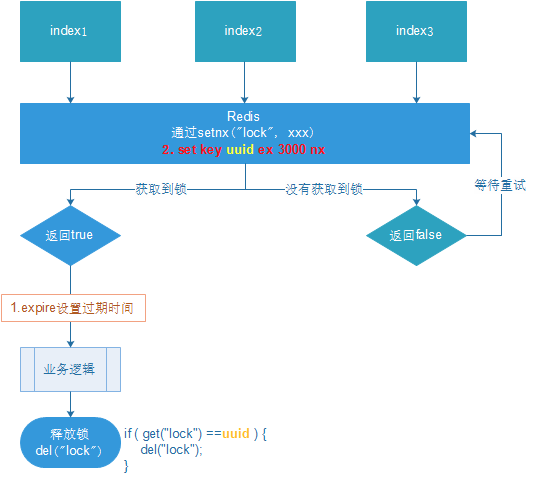
EX second ：设置键的过期时间为 second 秒。

PX millisecond ：设置键的过期时间为 millisecond 毫秒。

NX ：只在键(key)不存在时，才对键(key)进行设置操作。

XX ：只在键(key)已经存在时，才对键(key)进行设置操作。

Redis SET命令用于设置给定key的值，如果key已经存在其他值，SET就会覆盖，且无视类型。



由于redis删除操作缺乏原子性，可以使用lua脚本进行删除，但是由于redis主从切换时，有误删锁的情况，所以不建议使用，后来咱们演变为redission框架进行分布式锁的实现，原理和redis的一样，redission给进行了封装，通过调用lock以及unlock方法进行加锁和解锁，使用更简单。

最终我们自定义了一个注解结合AOP+redission框架的方式，把这个分布式锁加以复用，需要用到分布式锁的地方可以直接打注解就可以了。

**有关于redis分布式锁的演变，你可以当做在开发中遇到的问题来讲。**

## 3. 商品搜索模块

### 3.1 业务参考话术：

商品搜索是根据关键字进行拼配，从已有的数据库中摘录出相关的记录反馈给用户，在我们项目中有两个搜索入口，一个是商品的三级分类还有个就是用户在搜索框中输入关键字。我们主要是采用ES来实现业务，在ES中我们主要存入了分类id与名称、skuId、价格、名称、图片、品牌的id、名称、logo的url、平台属性数据以及热度排名字段等内容，我们把商品名称使用ik分词器进行了中文分词，同时还可以通过平台属性值、分类、价格区间以及品牌等作为搜索过滤条件，给用户展示商品名称、价格和默认图片等信息。

### 3.2 模块相关问题：

#### 1）ES的倒排索引？

正排索引：根据id去找到相对应的词

|  |  |
| --- | --- |
| Id（索引） | 词 |
| 1 | 红海行动 |
| 2 | 红海事件 |
| 3 | 红海行动事件 |
| 4 | 湄公河行动 |

倒排索引：根据词，看这个词都在哪个id（索引）里出现了

|  |  |
| --- | --- |
| 词 | Id |
| 红海 行动 | 红海1、2、3；行动1、3、4 |
| 红海 事件 | 红海1、2、3；事件2、3 |
| 红海 行动 事件 | 红海1、2、3；行动1、3、4；事件2、3 |
| 湄公河 行动 | 湄公河4；行动1、3、4 |

#### 2）ES过滤条件有哪些？

分类信息、平台属性、品牌、价格区间、热度、评论数等

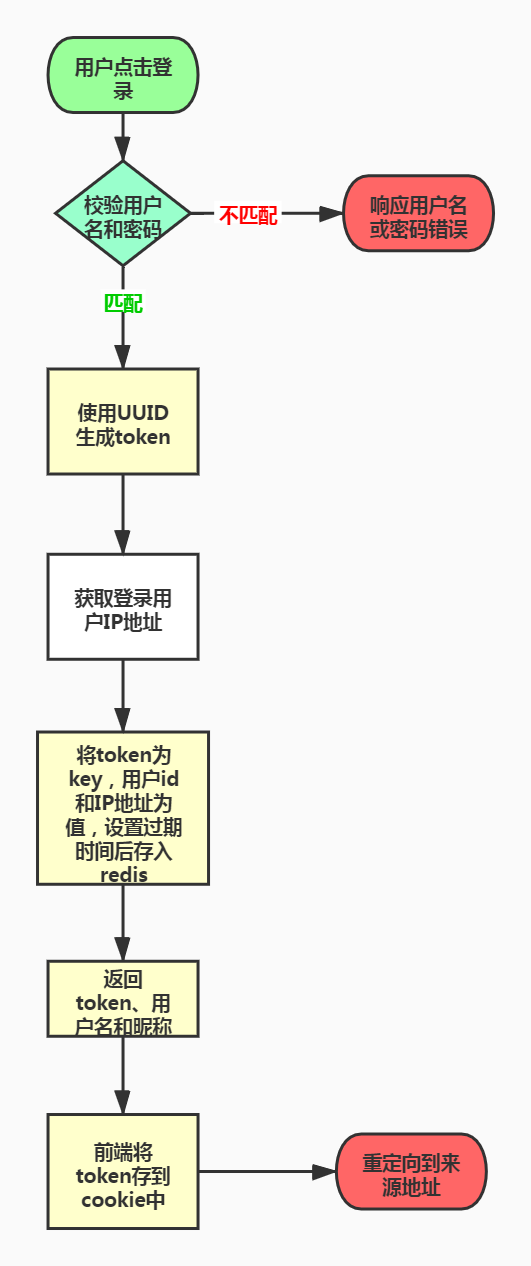
#### 3）ES数据同步问题怎么处理？

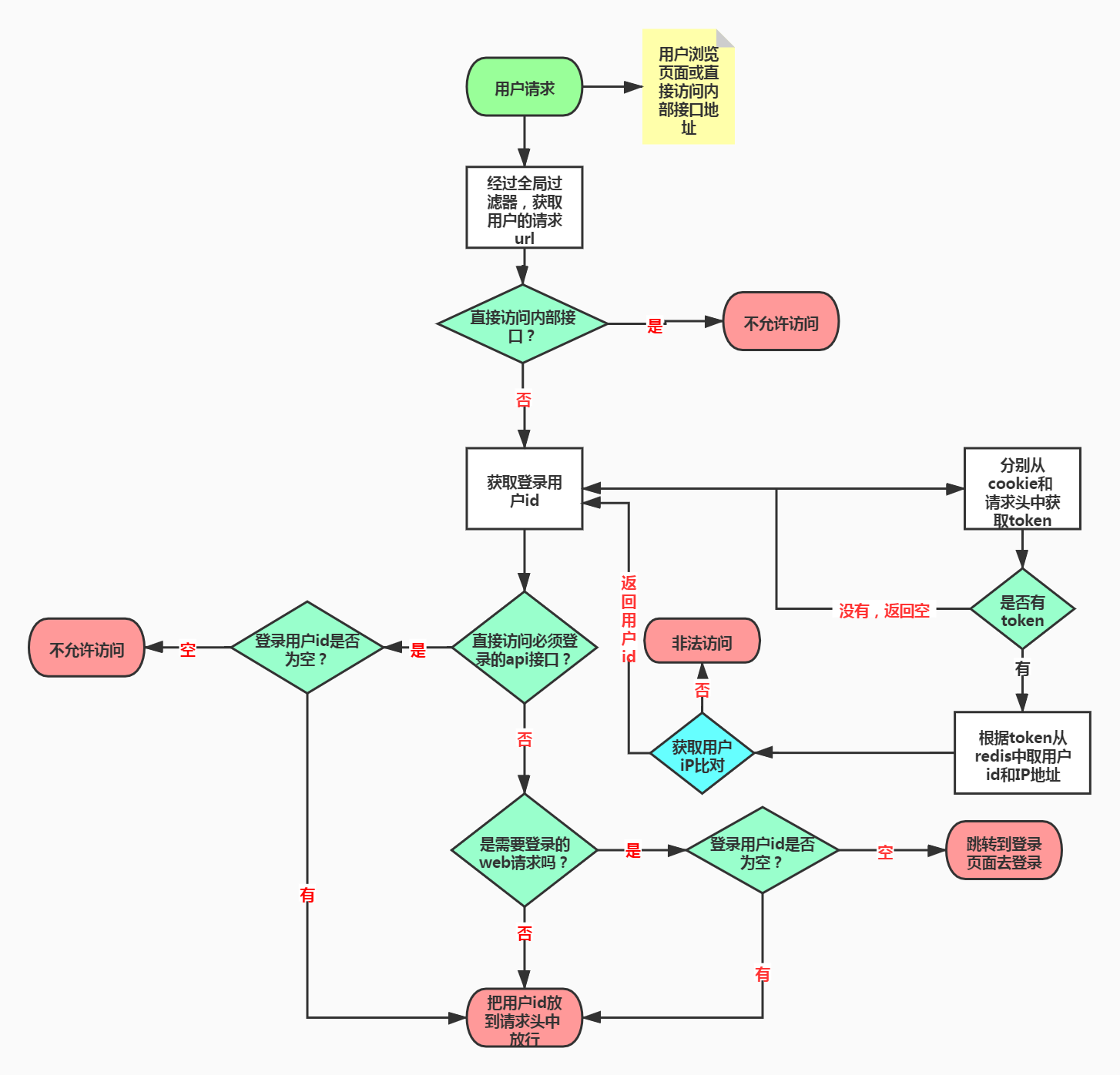
当商品上架时，发送mq消息通知搜索服务，当搜索服务监听到消息后把商品数据加入ES中，同样，当商品下架时也是通过mq进行消息传输来实现数据的最终一致。

## 4. 单点登录模块

### 4.1 业务参考话术：

单点登录就是访问项目时，用户只需要登录一次，就可以去访问所以内容。我们是采用网关全局过滤器、token认证以及redis来实现， 登录业务，用接收的用户名密码核对后台数据库，核对通过，用uuid生成token，将用户id加载到写入redis，redis的key为token，value为用户id。登录成功返回token与用户名和昵称，将token与用户信息记录到cookie里面重定向用户到之前的来源地址。用户再次访问其他需要登录的内容时通过网关过滤器进行拦截校验。具体业务参考以下流程图：





### 4.2 模块相关问题：

#### 1）cookie被禁用了能登录吗？怎么解决的？

不能登录，因为token存在cookie中，解决办法是参考京东的给用户提示：请您启用浏览器Cookie功能或更换浏览器

#### 2）怎么防止cookie盗用（盗链）？

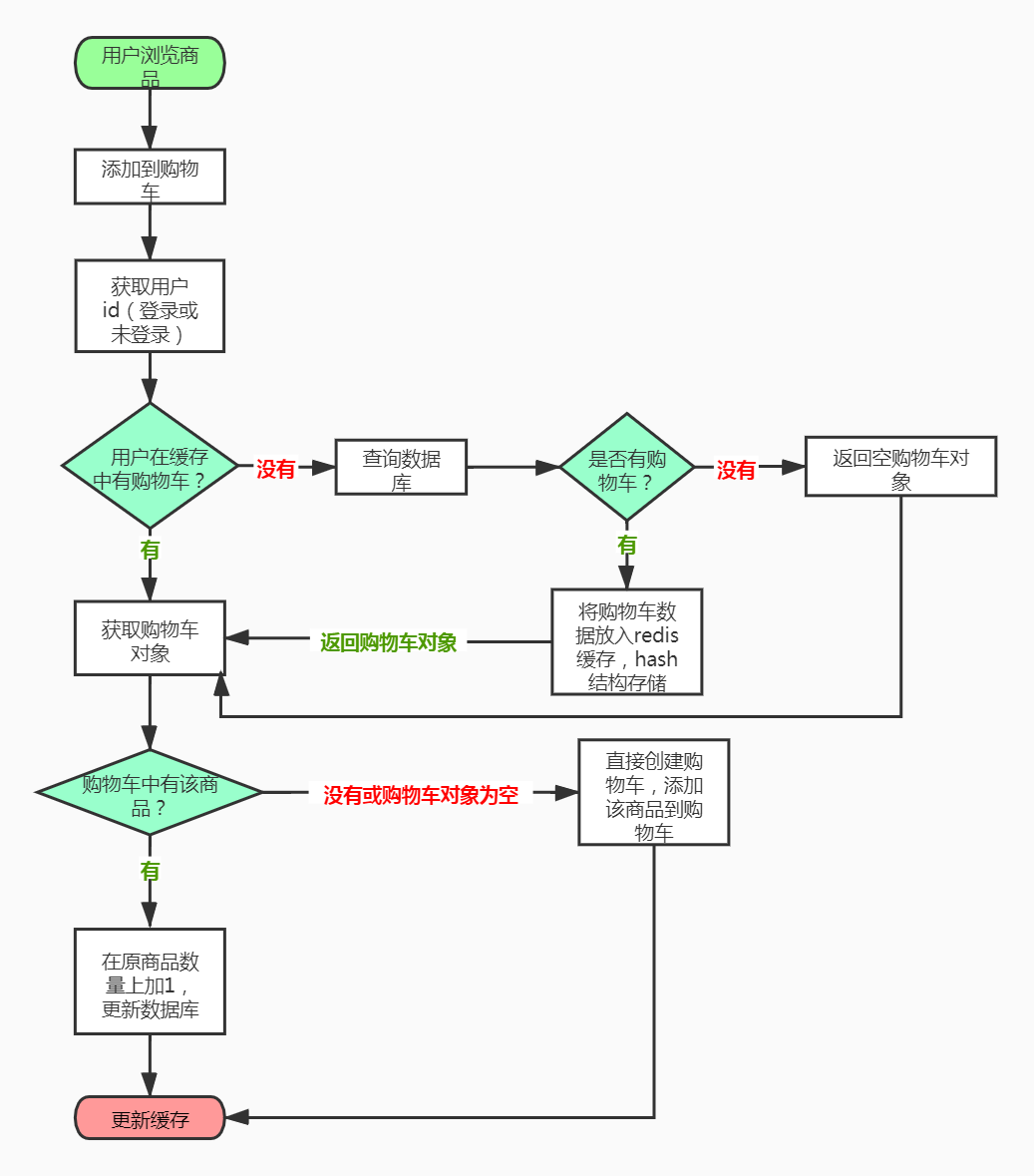
采用https进行加密传输，或者在token中加入以用户访问ip作为秘钥，解密token比对用户访问ip地址，ip不匹配，重新登录。

## 5. 购物车模块

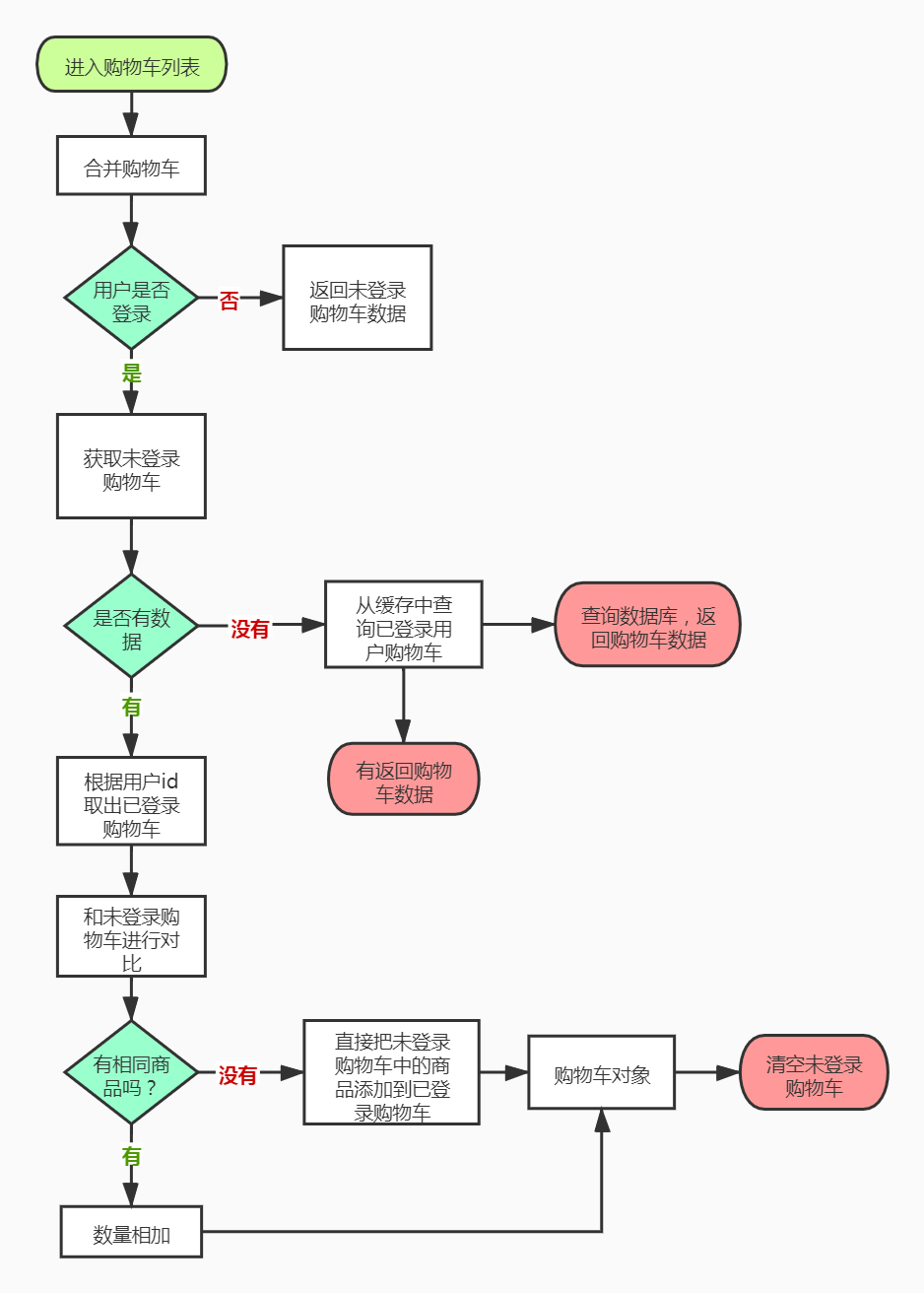
### 5.1 业务参考话术：

我们项目对于购物车的设计主要分为两个部分：用户未登录和已登录两种状态。在添加购物车时需要进行判断用户是否登录，如果未登录，则将购物车的信息放在redis中，采用hash结构存储，key为uuid（用户的临时id）随机生成，field为skuId，value是商品购物项信息，同时在数据库进行持久化操作，将reids的Key（临时用户id）存放到cookie中。用户已登录状态下，cookie中会存放token，根据token我们能获取到用户的id，所以采用userId作为hash结构的key，field为skuId，value是商品购物项信息。在进入购物列表时，会判断用户的登录状态，如果是未登录的话直接返回未登录购物车数据，如果是已登录，就判断是否存在未登录购物车，存在就进行购物车合并，然后再删除未登录购物车数据。

添加购物车流程：



合并购物车流程：



### 5.2 模块相关问题：

#### 1）购物车未登录有过期时间吗？已登录后有吗？

未登录的话：7天或15天

淘宝的设计是没有未登录购物车（如果面试不好说，咱们也可以说这种模式）

已登录情况下是永久存储，reids的数据有过期，主要是释放 redis’中数据，数据库采用永久存储。

#### 2）购物车能添加多少个商品？

49件或者99件商品，只要合理就行

#### 3）每个商品最多能加多少件？

库存充足的情况下可以添加99个，京东是200个

#### 4）添加购物车减库存吗？

添加购物车不减库存，验库存数量

#### 5）进入购物车列表的时候验价格吗？

验价格，如果价格有变化，展示新价格，给用户提示

#### 6）购物车里的商品价格变换了有提示吗？库存无货了有提示吗？

因为进入购物车列表时进行库存和价格的验证，所以会有提示。

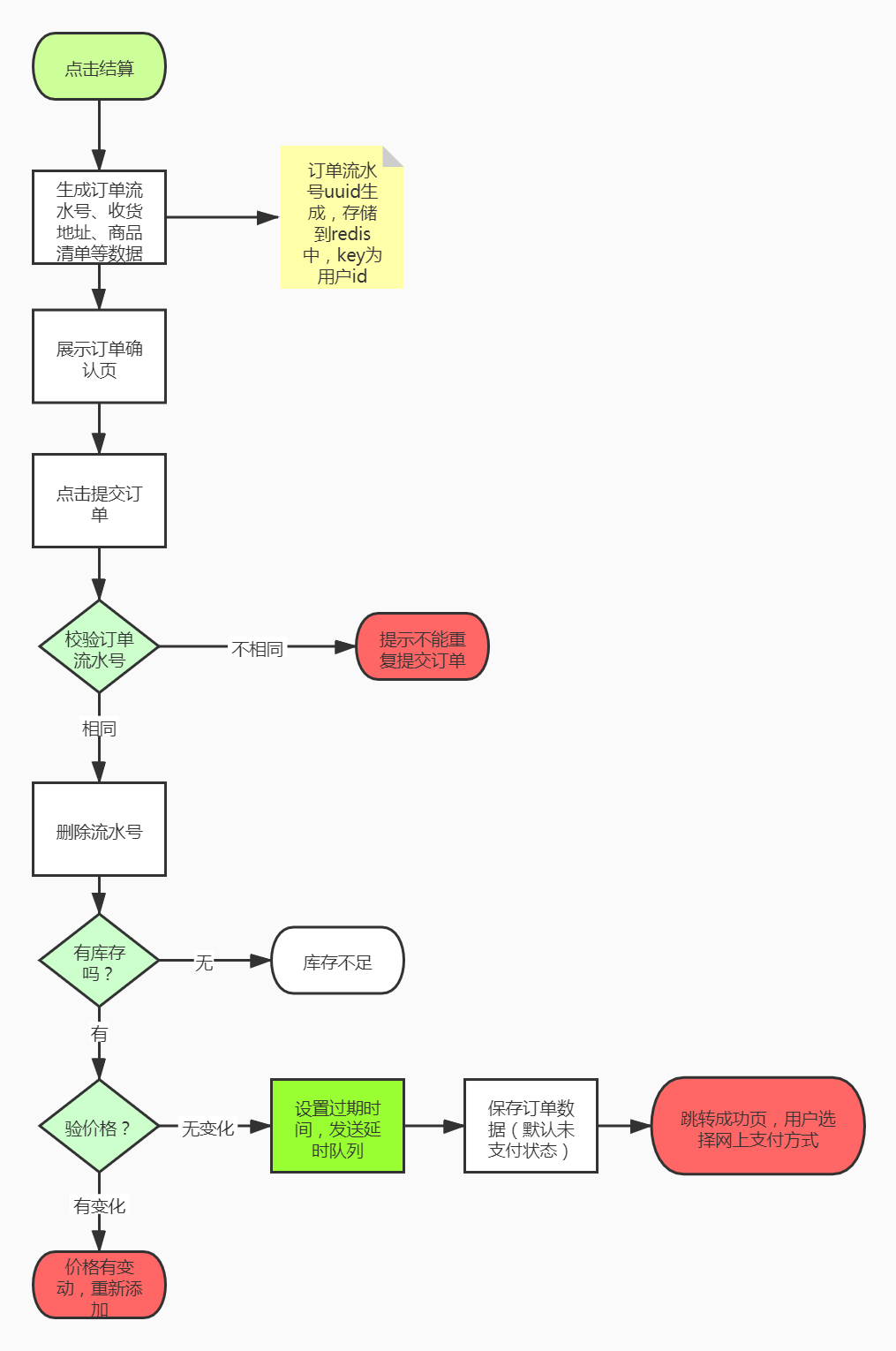
## 6. 订单微服务：

### 6.1 业务参考话术：

整个订单模块有结算页、下单、对接支付服务和对接库存管理系统等功能，当用户发起结算请求时，由于用户在没有登录的情况下也可以点击结算，所以我们要先判断用户是否登录，只有已登录的情况下，才能跳转到结算页面，在结算页，用户可以选择收货地址，给用户展示订单信息，用户选择支付方式，提供了微信支付和支付宝支付，确认订单信息，然后提交数据到后台，生成对应的订单表、订单详情表和订单物流表（当订单生成的时候，我们要调用对应的库存系统针对订单的商品数量进行验库存，还要进行验价格）。当订单创建成功之后，自动跳转到成功页面（将订单数据和到期时间传递过去）。

这块我们设置的订单的有效时间为24小时（这个时间可以自己定，只要合理就行），因为我们利用延时队列实现定时消息发送，消费者到时间后监听到消息，进行订单校验，如果订单是未支付状态，把订单状态修改为关闭订单。

订单的主要状态有 未支付、已支付、待发货、已收货、待评价、已完成、已关闭等。



### 6.2 模块相关问题：

#### 1）订单有效期是多久，怎么取消订单？

订单有效期为24小时，我们采用rabbitmq的延时队列，在生成订单时，发送延时消息，设置消息24小时后触发，然后监听到这个消息后，进行订单的取消。

#### 2）怎么防止订单重复提交？

在进入结算页面是我们生产了一个流水号，然后保存到结算页面的隐藏元素中一份，Redis中存一份，每次用户提交订单时都检查reids中流水号与页面提交的是否相符，如果相等可以提交，当订单保存以后把后台的流水号删除掉。那么第二次用户用同一个页面提交的话流水号就会匹配失败，无法重复保存订单。

#### 3）订单超卖问题怎么解决的？（库存只剩1件商品，多个用户同时下单）

我们采用的是下订单不减库存，只验证库存，在支付成功后再去扣减库存，如果库存扣减失败，通知后台进行补货，如果这个商品不能补货，人工客户介入，和买家进行沟通，给予退款或相应补偿。

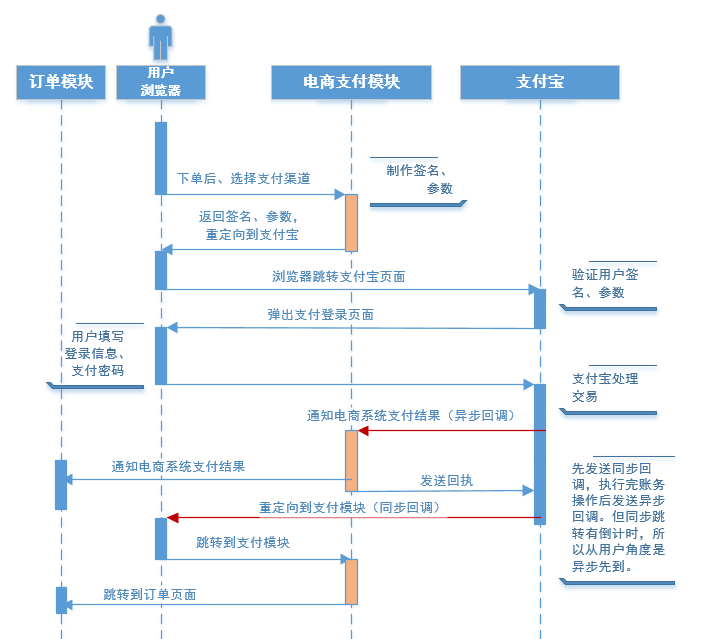
如果想实现不超卖，就得在下单时进行库存锁定，（可以使用数据库锁方式）然后减库存操作。

## 7. 支付微服务

### 7.1 业务参考话术：

当用户发起支付请求时，我们采用支付宝或者微信支付，以支付宝为例，我们要制作调用支付宝统一下单接口（支付接口）需要的签名和参数，我们先封装了一个AlipayClient配置类，把商户appid，支付宝公钥，开发者私钥，统一下单接口url以及签名方式等默认参数进行传入，从而生成AlipayClient对象交于spring管理。当调用支付时，我们会把订单交易标号、商品码，交易金额以及同步回调地址和异步回调地址等参数传给支付宝，从而生成支付页面，在这个过程中我们会记录支付信息存储到数据库中，当用户扫码支付后，支付宝回给用户跳转页面，主要是我们设定的同步回调地址。平台这块是根据支付宝的异步回调来判断用户是否支付成功，接收到异步回调时我们会验证签名，以校验合法性，同时会使用rabbitmq给订单模块发送一个校验成功的消息，以便于订单模块的后续处理。

支付宝还提供了退款、对账、关闭交易等接口，在项目中也都使用了。



### 7.2 模块相关问题：

#### 1）支付宝支付需要的参数？

主要参数有appid、开发者私钥、支付宝公钥、支付网关地址，同步回调url、异步回调url、请求数据格式、编码方式、签名方式、订单交易编号、订单金额、商品编码等

#### 2）支付锁库存吗？

我们做的是支付不锁库存，只有在支付成功后，去锁定库存进行响应的扣减。这块面试官会说超卖了，可以参考上面订单超卖内容进行解答。

#### 3）支付日志表（信息表）中记录了什么东西? 有什么作用？

主要记录 支付类型、金额、交易号、订单编号、交易内容、支付状态和时间等。

支付日志表主要是记录支付的状态转换和数据以便于和支付宝微信等三方平台进行对账等。

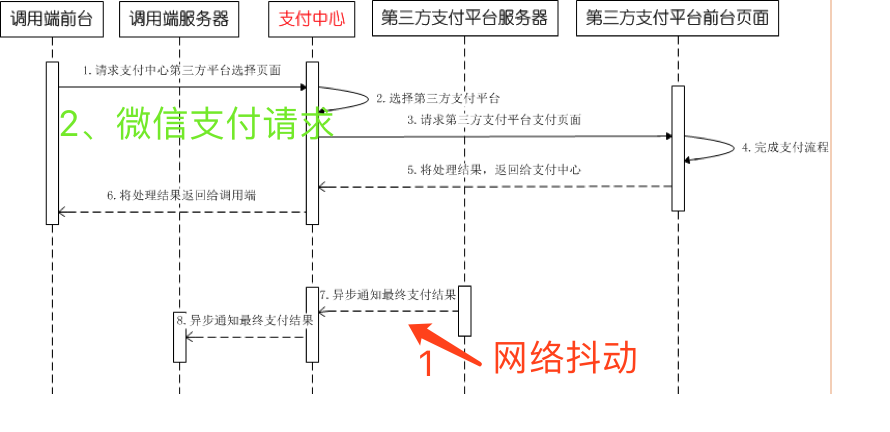
#### 4）重复支付问题？

##### 场景1

对同一个支付平台web端和app端的重复支付，同一个支付平台针对同一笔业务订单生成的流水号是唯一的，第三方支付平台对同一个流水号只进行一次支付，不会重复支付。

##### 场景2

例如银行的扣款成功的通知结果延迟了，在同一笔业务订单因为没有收到成功的结果通知，发生第二次支付并正常成功后，才收到第一次支付成功的通知。如下图所示

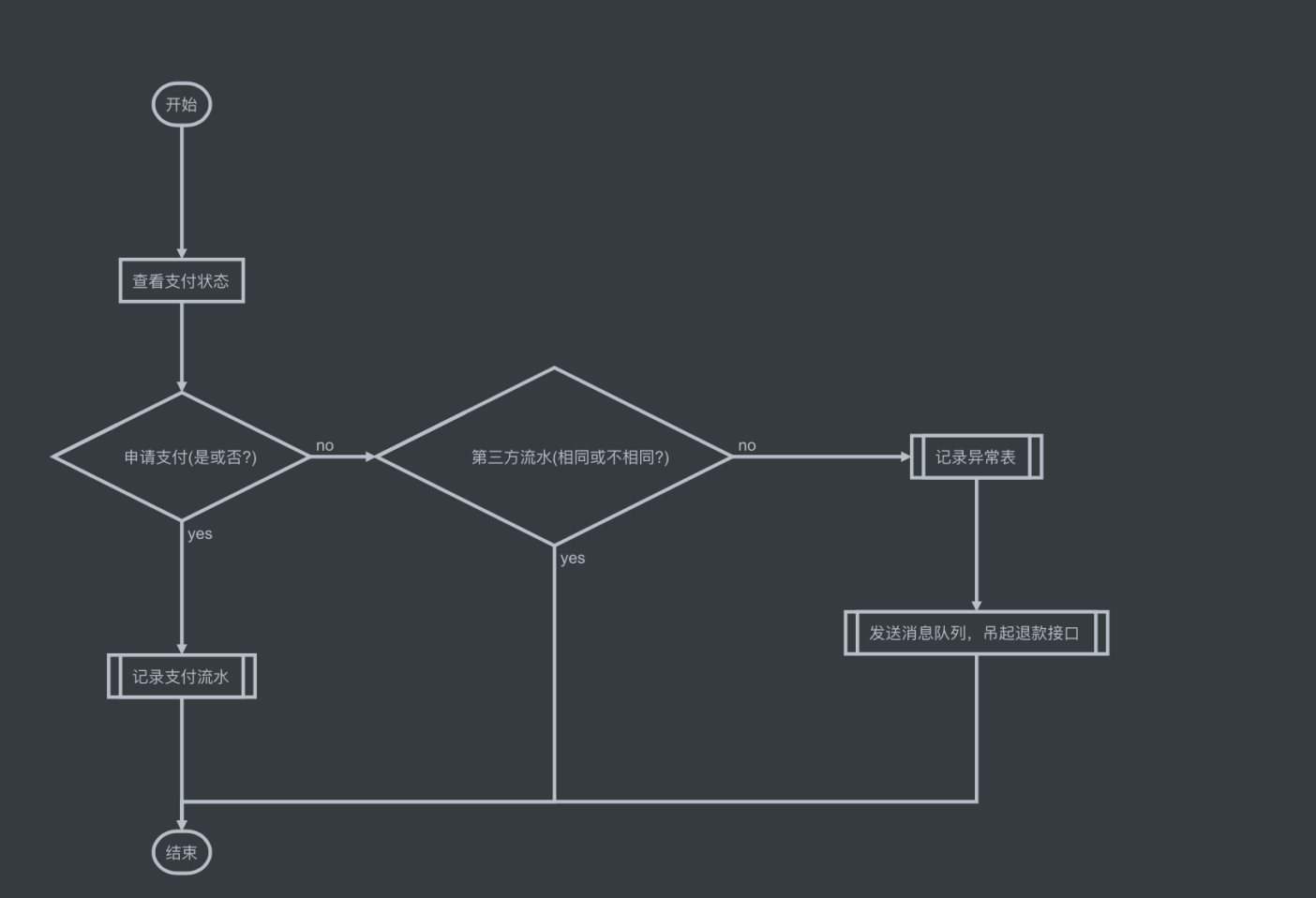


##### 解决方案

在第三方支付中心异步回调时，实现以下功能：

查询支付中心流水表，按订单号查询支付状态是否为申请支付状态，如果是，将此订单支付状态修改为支付完成，记录到支付中心流水表；如果不是申请支付状态，对比支付中心流水表中现有记录的第三方平台交易流水号和第三方平台返回的第三方平台交易流水号，如果两者都相同则不进行任何操作，表示是第三方支付的重复通知，直接返回结果即可；如果两者不同，则记录到异常支付单表中。需要运营人员进行线下退款处理或者直接调用退款接口退款。

流程图如下：



## 8. 秒杀微服务

使用下面的流程图对比代码，多去看几遍，熟悉了就知道秒杀的解决方案了，需要理解。

